TAPR

ARRL

21st

$20.00

ARRL and TAPR DIGITAL COMMUNICATIONS CONFERENCE

DENVER, COLORADO

September 13-15, 2002

21st ARRL and TAPR DIGITAL COMMUNICATIONS CONFERENCE

ARRL
225 Main Street
Newington, CT 06111-1494 USA
tel: 860-594-0200 www.arrl.org

Tucson Amateur Packet Radio
8987-309 E Tanque Verde Rd #337
Tucson, AZ 85749-9399 USA
tel: 972-671-8277 www.tapr.org

Printed in USA

ISBN: 0-87259-875-6

ARRL Order Number: 8756

First Edition

Welcome!

The 21st ARRL and TAPR Digital Communication Conference is fertile ground for the kind of technological innovation that has always been a hallmark of Amateur Radio.

In these proceedings, you will find several examples of digital innovation at its best. APRS continues to flourish and a number of papers reflect advances made in this area. Other presentations describe clever ideas ranging from proposals to integrate simultaneous voice and data repeater channels, to three-dimensional graphic tools to enhance amateur communication.

The hot topic at this year's "Sunday Seminar" was *SDR*—Software Defined Radio. Many believe that SDR is the future of Amateur Radio. Read the SDR presentations in his book and see for yourself.

As always, I hope these proceedings and this conference will inspire you to begin some digital explorations of your own—and that we'll have the pleasure of seeing one of your articles in the 2003 edition.

David Sumner, K1ZZ
ARRL Executive Vice President

September 2002

Call for Nominations: New ARRL Technical Awards

Nominations now open for an exciting new slate of national ARRL awards recognizing service, innovation and microwave development in the technical arena!

The ARRL Board of Directors created a program of technical awards to recognize service, innovation and microwave development. In taking the action, the Board noted the international regulations recognize the "technical investigations" conducted by the amateur service. In our own domestic rules, the FCC recognizes the amateur's "proven ability to contribute to the advancement of the radio art," the need to promote technical skills, and expand the "existing reservoir . . . of trained operators, technicians and electronics experts." The Board also noted that one of the purposes of the League is to promote experimentation and education in the field of electronic communication, research and development, and the dissemination of technical, educational and scientific information.

All of the above is fundamental to the amateur service and deserve recognition and encouragement, hence the Board's initiative to recognize achievement in technical service, innovation and microwave development. Now is the time to nominate your colleagues for one or all of the awards described below!

ARRL Technical Service Award

The Technical Service Award is given annually to the licensed radio amateur whose service to the amateur community and/or society at large is of the most exemplary nature within the framework of Amateur Radio technical activities, including, but not limited to, the following:

- Participation or leadership in technically-oriented organizational affairs at the local or national level;
- Service as official ARRL volunteer; i.e., Technical Advisor, Technical Coordinator, Technical Specialist.
- Sharing of technical education and achievements with others through articles in the Amateur Radio literature, and at club meetings, hamfests and conventions; and encourage others to do the same.
- Promotion of technical advances and experimentation at vhf/uhf and with specialized modes, and work with enthusiasts in these fields.
- Service as technical advisor to clubs that sponsor classes for obtaining amateur licenses or upgraded licenses.

- In times of emergency or disaster, provide technical expertise to Amateur Radio service providers, government and relief agencies to assist with the establishment of emergency communications networks.

- Referral of amateurs in the section who need specialized technical advice to appropriate sources.

- Work with local clubs to develop RFI/TVI committees in the section for the purpose of rendering technical assistance as needed.

- Assistance to local technical program committees in arranging suitable programs for ARRL hamfests and conventions.

Formal nominations may be made by any ARRL member. Supporting information, including the endorsement of ARRL affiliated clubs and elected or appointed League officials, should be submitted along with the nomination document. The ARRL's Future Systems Committee will serve as the award panel and will review the nominations received from the members and select the winner. The winner will receive a suitably engraved plaque, and travel and accommodations expenses to enable him or her to attend an ARRL convention at which a formal presentation will be made.

Nominations should document as thoroughly as possible the record of technical service of the nominee during the previous calendar year. Additional information concerning the character of the nominee should be as complete as possible. Nominations should be sent to: ARRL Technical Awards, 225 Main St., Newington, CT 06111.

The ARRL Technical Service Award is intended to provide encouragement, and a tangible reward, for amateurs who are outstanding in the field of technical service. It also provides an opportunity for Amateur Radio, and its many benefits for society, to be brought to the attention of the public. Nominations must be received at Headquarters by March 31.

ARRL Technical Innovation Award

The amateur community has witnessed great changes over the past 75 years, from spark to space. At the heart of many advances in the radio art has been the amateur himself. It has been and will remain ARRL policy to encourage amateurs to continue to lead at the forefront of technological advancement as recognized by the Board. The ARRL Technical Innovation Award is granted annually to the licensed radio amateur whose accomplishments and contributions are of the most exemplary nature within the framework of technical research, development and application of new ideas and future systems in the context of Amateur Radio activities, including but not limited to:

- Increasing use and development of higher-speed modems and improved packet radio protocols;

- Expanded use of personal computers in Amateur Radio applications;
- Increased efficiency in use of spectrum;
- Alleviation of long-standing technical problems, e.g., antenna restriction, competition for spectrum, EMC;
- Digital voice experimentation;
- Improved portable and mobile communications capability through Amateur satellite development;
- Increased use of solar, natural and alternative power sources for field applications;
- Development of practical compressed video transmission systems;
- Spread-spectrum technology and applications; and
- Development of assistive/adaptive technology for disabled amateurs.

See above for information on how to nominate. The winner will receive a cash award of $500, a suitably engraved plaque, and travel and accommodations expenses to enable him or her to attend an ARRL convention at which a formal presentation will be made.

Nominations must be received at Headquarters by March 31.

ARRL Microwave Development Award

A great frontier for amateurs is the microwave bands. With room to move, the microwave region of our spectrum presents amateurs with a vast test bench for new modes, as well as development of traditional ones. The ARRL Microwave Development Award is given each year to the amateur (individual or group) whose accomplishments and contributions are of the most exemplary nature within the framework of microwave development, i.e., research and application of new and refined uses and activity in the amateur microwave bands, along with the millimetric bands above 30 GHz. This includes adaptation of new modes both in terrestrial formats and satellite techniques.

See above for information on how to nominate. The winner will receive a suitably engraved plaque, and travel and accommodations expenses to enable him or her to attend an ARRL convention at which a formal presentation will be made.

Nominations must be received at Headquarters by March 31.

Table of Contents

1-Wire APRS Weather Station

William Beals NØXGA
15014 East Idaho Place
Aurora, CO 80012
will@beals5.com

Russell Chadwick KBØTVJ
4371 North 63rd Street
Boulder, CO 80301
russ@wxqa.com

Abstract

This project and resulting TAPR kit implement the necessary hardware and software to sense and display data from various low-cost 1-Wire weather sensors. With the addition of a TNC and radio, it forms a complete, stand-alone APRS-compatible weather station. Weather information is also available on a 4-line by 20-character LCD. The project is designed to be easily extensible and upgradeable—even for applications that are not weather related.

Key Words

One-Wire, 1-Wire, APRS, Weather, Peet, Dallas Semiconductor, Motorola, LCD, temperature, wind direction, wind speed, humidity.

Introduction

After much research, the authors came to the conclusion that there wasn't quite the "perfect" weather station intended for stand-alone APRS use. Our definition of "perfect" had the following requirements. 1) Affordable 2) Reproducible 3) Expandable 4) Accurate 5) No computer required 6) Low power and single supply, and 7) APRS compatible. Multiple consumer-grade weather stations exist, but in order to get on the APRS airwaves required either a dedicated computer or special "logging software" to transmit a non-APRS format. We had no convenient "buy it" option available. We considered several reputable projects where sensors were made out of household parts, but they relied on surplus parts that failed our second criterion, reproducibility. About that time, Dallas Semiconductor introduced a line of sensors called 1-Wire devices that required only two wires (marketing folk don't count ground I guess). To promote these parts they introduced a wind/temperature sensor based on these chips. With a decent and affordable sensor in hand, a micro chosen, and some welcome TAPR support, the project was born.

Project History

The project started on a 6805 microcontroller. Based on personal preference, all of the source code was written from the ground up, starting with and referring to little else beyond the Dallas chip specifications. The only times I "cheated" and looked at other code was referencing some C

code for the CRC checkers. This clearly had drawbacks in that I had to learn several basic lessons the hard way, however by starting from scratch, I had significant flexibility in the overall code architecture and chose one best suited for this project.

During the early project evolution, several major changes took place. The micro was upgraded to a Motorola 6808, the external serial EEPROM functionality was incorporated into the 6808 internal flash, and we even changed revisions of the 6808, from a 20Kbyte part to a 32Kbyte part. At that point the printed circuit board was created, so only software changes were allowed.

Even after the hardware platform became fixed, several significant features were added and upgraded. Most notable were adding a software downloader, overhauling the code for purely interrupt-driven data gathering, and the adding of several weather sensors as well as a time reference. All these upgrades have been available free to anyone who purchased the weather station or constructed it from scratch. Starting fairly early, the full source code has also been available for download as well.

The Hardware

The 6808 requires very little in the way of external support components for normal operation. These include a 5V regulator, crystal oscillator (not really required), the one-wire interface, and the UART buffers. Additional circuitry is included for program debug, this includes an extra crystal oscillator, additional UART interface, and a slightly unusual reset circuit.

The need to put the debug circuitry on every "production" board was heavily debated during the board design phase. This is a very general purpose board and we expect projects other than the weather station to be possible with this hardware platform. To support this, it made sense to enable as many people as possible to be developers.

Weather Data

During the initial stages of the project, the official APRS data formats were in the final stages of getting defined by the APRS working group. The final version I used was version 1.01. The only area were the station is in a gray zone is in the definition of the station type. The station identifier field calls out a one-letter software type followed by a multi-character weather unit type. For the software type, the specification defined six categories of equipment based mostly on the operating system: Windows, DOS, Mac, and Linux. A unique letter identifies each of these categories. Unfortunately, the weather station fit none of these categories. I arbitrarily chose a seventh identifier letter "e" for Embedded or Experimental as it was an unused letter. The weather unit type characters do not need to follow any format, so I chose "1w" for one-wire. Hence the station identifier is "e1w".

Both positioned and positionless weather data formats are supported. If the user specifies their coordinates in a setup screen the positioned format is chosen.

Data transmitted by the weather station clearly depends on the sensors installed. Sensors currently supported are wind speed, wind direction, temperature, rain, and humidity. In addition to the basic weather data from those sensors, calculated data such as dewpoint is displayed but

not transmitted. For rain, the intervals reported on APRS are rain last hour andsince midnight. In addition, rain for month-to-date, and "since-user-reset" are displayed on the LCD. Wind speed is measured every 5 seconds. Average wind speed over 5 minutes is reported for each APRS packet along with the peak gust measured over the 5-second sampling time.

Daily stats are also displayed for the current as well as previous day. This includes high temperature, low temperature, peak gust, and rainfall.

This is only the current sampling of weather data gathered and reported. This project by its nature is easily expandable. The most noticeably absent sensor is barometric pressure. The authors are waiting for a suitable mass producible sensor to be available before adding the support for it. There is a decent chance such a sensor will be available before this article gets published.

Peet Mode

In addition to APRS-formatted data, a mode exists to send data out of the unit using Peet format. There was a specific user request to support some existing equipment that understood only Peet formatted data. When in this mode, only raw (unaveraged) data is sent every five seconds.

Reporting accurate weather data

For a project such as this, being able to get, process and present the data is only half the battle. The other half of the battle is to do so as accurately as possible. Russ brought his expertise in this area, providing the most accurate algorithms possible and practical for this project. Weather data is inherently very "noisy" data. To sample this raw data every five minutes and broadcast that one sample does not give a good representation of the current weather conditions. Additionally, simple averaging is not practical, especially for information such as wind direction.

Download Monitor

Early in the project the main processor was changed from a UV-erasable EPROM-based 6805 micro to a Flash EEPROM-based 6808 micro. Once the main micro became flash-based, that brought the possibility of allowing users to upgrade their code without having to remove the micro from the main board. It would further allow this upgrade without needing the full debugger toolset. This quickly became a critical feature of the design and has proven to be a great asset during the projects lifetime, allowing users to easily upgrade to newer versions of the software without requiring anything more than a PC and a simple DOS-based program.

The Flash memory inside the micro is divided into three regions. The first is the boot area. This is never altered during normal operations. The second is the user-data area. This area stores user data and preferences. It is preserved during code updates. The final area is the application area. This is the area that is erased and update during a code update.

The boot area resides in the boot vector space of the flash and assumes control after reset. It occupies 2K of the flash space. After reset, this code checks to see if the up and down buttons are being pressed simultaneously. If not, the boot controller passes control to the application area. If both buttons are pressed, the boot loader initializes the display and presents a message

saying it is ready for a download. It then sends out a query character, waiting for the PC to start a download. As soon as this character is recognized and acknowledged, the boot loader erases the application area and requests new application code from the PC until the new application is loaded.

The user data area is where sensor IDs are stored as well as any other permanent information such as position string, UTC offset, etc. It is also approximately 2K in size. A significant advantage of the user-download feature is that this data is NOT erased when new application is loaded, hence preserving any previous settings. The development tools can only erase the entire flash including the user settings. A default of 0xff is required for any unused data, allowing a known value to be present for any new variables that become defined with a new software download.

The application area is the remainder of Flash, approximately 28K in size. This area is not divided any further, if there is a code update, this entire area is erased and reprogrammed. Since inception, there have been nearly 30 releases to both introduce significant new features as well as fix numerous coding bugs.

An Upgradeable Project

As evidence of the usefulness of the upgrades, the initial release of the weather station only included basic monitoring of wind direction, speed, and temperature. During the nearly thirty releases, the following features were added either as pure software enhancements, or additional sensor support for which the sensor just needed to be added to the 1-wire network.

Computer (non-averaged) data mode
Updated calibration constants for wind speed
Added error counters screen
Made all data gathering routines interrupt-driven
Wind direction optionally, N, NW, vs. just degrees
Added today/yesterday stats screen
Added Peet mode support
Added rain gauge support
Added support for better DS18S20 temp sensor
Added support for TNCs with LTP command
Added positioned weather format
Added metric display screens
Added support for DS1994 real-time clock
Added full baud rate support in all modes
Added bus voltage monitoring (great for debug!)
Added support for humidity sensor
Added support for new AAG wind/temp sensor
Added ability to "remove" a sensor
Added external radio power control (external HW required)
Added battery voltage monitoring (external HW required)
Added dewpoint option vs. humidity only

This list only includes new features, not the numerous bugs that were discovered while adding these features. Also, in many cases the features were requests from users who purchased the weather sensor and placed requests for additional features after using the unit for a while.

Single-supply design

(A small digression here) A significant drawback with PC-based weather stations in this authors opinion is power consumption. Assuming an average PC draws approximately 200 watts (even more if the monitor is on), power consumption alone adds up to a tangible impact to the electricity bill. If you are curious, multiply 146.4 (the number of kilowatt hours in a month of 24/7 usage at 200 watts) times your local utilities cost per kilowatt hour. For our local utility, this adds up to about 11 dollars a month! Naturally, your numbers may vary—especially if your 24/7 computer is a laptop or some other watt-miserly machine. This ongoing cost was a major reason for wanting to expend the effort for an embedded weather station instead of just using one of the many excellent PC-based solutions that were available at the time.

Fully configured with a standard setup and some margin, the weather station draws a maximum of four watts (12V at 300ma or 22 cents per month). Almost half this power is for the LCD backlight, so if you leave it off, power is even further reduced. This is sufficient for any domestic use. The choice to use a 12V supply even though only 5V is needed is that this is already a very common power voltage in almost any ham shack. Also, for a remote situation 12V is also a very common DC supply.

During the design we debated over the type of regulator to use to get the 5V supply from the 12 rail. The quick and dirty linear regulator was certainly the easiest and cheapest. For a remote location that might be solar powered, a switching regulator would clearly have made more sense. We ended up choosing the cheaper solution that wasted some power in recognition that most of the station would be AC-powered. For anyone running from solar power, they could make the choice to use the existing linear or install a switching regulator in the patch area.

Future Work

The only major sensor still missing is barometric pressure. This sensor has proven to be somewhat problematic to make cheaply and accurate at the same time. The author also really wanted to support a sensor that was produced by either Dallas Semi or AAG, but neither company has offered one as of this writing. A design from the amateur (not just radio) community is gathering momentum and support for this sensor may exist about the time of this papers publication.

Other future work includes using more of the DS1994 features besides the real-time clock. The part also has a 4Kbit non-volatile memory. It would be nice to log daily statistics to that memory in a format that would be readable by any PC running Dallas Semiconductors TMEX file system. This would allow logging up to approximately one month of data and then be able to transfer these logs to a PC in a very convenient manner.

There is also nothing required other than programming time to be able to support parsing of incoming APRS data and displaying it on the screen. The numerous formats, exceptions, and

variations of incoming data would be a formidable chore and so far continues to be avoided by the author.

Finally, one of the original intents of the hardware design was to be a general-purpose board for projects besides weather stations. So far no other projects have materialized. Hopefully some day this will happen.

Conclusion

It is a rare opportunity in the hobbyist community to not only want to work on a project but also find a group of people and an association that is willing to help you in that process. This weather station is such a project. It would be dangerous to try to mention the numerous folk that helped with this project either directly in material ways or the hundreds of emails on the weather reflector. TAPR clearly deserves thanks as it provided the forum for all of the helpful people to work through.

References

Ian Wade, Editor. (2000) APRS Protocol Reference. The APRS Working Group

Dallas Semiconductor. DS18S20 High Precision 1-Wire Digital Thermometer
Dallas Semiconductor. DS1994 4Kbit Plus Time Memory iButton
Dallas Semiconductor. DS2401 Silicon Serial Number
Dallas Semiconductor. DS2407 Dual Addressable Switch Plus 1K–Bit Memory
Dallas Semiconductor. DS2422/DS2423 1K/4K–Bit 1–Wire RAM with Counters
Dallas Semiconductor. DS2438 Smart Battery Monitor
Dallas Semiconductor. DS2450 1-WireTM Quad A/D Converter

Author

Richard E Carter – KE1EV (ke1ev@arrl.net)

Abstract

This paper describes the PIC-E-II project. It includes information gathered from various sources or researched directly using test equipment. From a high-level, it describes what a packet is, what a PIC is, and how the PIC-E-II works.

Keywords

PIC, NMEA, APRS, AX.25, packet.

Overview

The PIC-E-II is an extension of the existing TAPR PIC-E kit. It shares the same modem chip and uses a processor chip from the same PIC family. The functional requirements of the project are significantly more ambitious however. The design attempts to extend the capabilities of a simple encoder to a self-contained APRS station. It is capable of not only transmitting a position, but of also receiving and processing APRS packets. System and software functional specifications are included as appendices at the end of this document. Consult these documents for details concerning system requirements.

The project has been supported with several volunteers. The hardware design has been a collaborative effort. We also cooperated in the selection of tools and development environment. A single software application is in process. It is being written by a single contributor, the author of this paper. Source is available on request. It is planned to provide open source for this software; restricting use to non-commercial projects. The software design for this application is described in this document.

Spare I/O ports are provided and a small breadboard area is planned for the board. Consideration was given to supporting inexpensive development tools, including a C compiler and an in-circuit debugger. The kit should be considered a potential breadboard for many amateur projects, not just those requiring AX.25.

I started this project because APRS holds a lot of promise for the marine community, but existing equipment doesn't address these needs. There is a strong need for a station that collects and transmits weather and position data collected from an NMEA bus. It should accept and store packets addressed to the station. It must be small and consume little power. It would also be useful if the station could monitor the ship's battery voltage, flood, fire and theft alarms; generating an emergency status message if parameters are out of limits.

I also think there is a need for a generic embedded processor kit that could be used for a variety of amateur radio applications. These applications need not be APRS related. In fact, then need not make use of the TNC interface provided by the kit.

Ax.25

The process of sending data over a radio using audio tones is described in the AX.25 specification. This transfer uses a network protocol. Industry uses a conceptual model to describe the basis for a network protocol implementation. This model describes the process in terms of network layers, sometimes called the ISO reference model. This model describes seven layers, three of which are used by APRS.

Physical Layer – Layer-1

At the physical layer, ax.25 is implemented using two audio tones, 1200hz and 2400hz. The bit rate is 1200 baud (833us/bit). This is suitable for 2m FM operation. Data is transmitted as eight-bit ASCII characters LSB first. During each bit period, one of the two tones is transmitted. The tone does not directly correspond to either a one or a zero. Instead, a change of tone indicates a zero bit and no change indicates a one. This is more easily explained using the illustration below.

The two tones are represented as A and B. If the byte being transmitted is 01110000b, the tones transmitted would be ABABBBA or BABAAAB (NB: bits are transmitted from right to left).

If a long series of sequential ones were transmitted, there would be no change of tone. Differences of the clock frequency of the transmitter and receiver stations could make reliable decoding of such a packet unreliable. To avoid problems caused by clock drift, the spec restricts transmissions to sequences of five or fewer sequential ones. This is accomplished using a process called bit-stuffing. Whenever five sequential ones are detected in the output data stream, a zero is stuffed in the sequence. This "stuffed" zero is removed by the receiver.

If the byte being transmitted is 01111110b, after stuffing it would be 010111110b. The tones transmitted would be AAAAAABBA or BBBBBBAAB. Notice that the eight bit transmission takes nine tone periods to send.

Data Link Layer – Layer-2

Data is transferred between host and client in frames. Each frame of data is called a packet, each of which is bound by a special flag marker. In order to initially synchronize the transmitter and receiver, this special flag is transmitted before the first data byte of the packet. This flag has six sequential ones bracketed by zeros (0x7E). Bit stuffing is not performed on this byte. This is the only normal condition where more than five sequential ones are found in a packet. If more than six sequential ones are detected at the receiver, the spec requires the receiver to abort the packet and search for the beginning of a new packet flag. A two byte CRC is appended to the end of the packet, followed by a flag. It is common practice for a transmitter to send a sequence of consecutive flags at the beginning of a packet. The transmitter may send a series of flags at the end also.

There is no acknowledgement defined by AX.25 at the data link layer.

Application Layer – Layer-7

APRS is an application that uses the AX.25 network protocol. The APRS spec defines the frame format for APRS as a UI frame (unnumbered) with nine fields of data. Some of these fields are really part of lower layers, but are described in the spec for completeness.

1. Flag – 0x7E
2. Destination – Callsign and SSID of the destination station.
3. Source – Callsign and SSID of the originating station.
4. Digipeater Address – variable number of digipeater station addresses.
5. Control – Defined as UI frame for APRS.
6. Protocol – No layer 3
7. Information – This field is implementation specific. The first byte defines the format of this field.
8. FCS – 16-bit CRC

There are many specific application formats for APRS data. The PIC-E-II transmits packets using the MIC-E[1] format. Since generic APRS messages have no particular destination address, the destination address field is unused. The MIC-E format makes use of this unused field to place some of the position data. The remaining part is included in the information field. There is an advantage to making packet length as short as possible, so the position information is packed to shorten the packet as much as possible. AX.25 restricts transmitted bytes to legal ASCII characters, so the packing algorithm is restricted to fewer than 256 codes per byte. This packing method is described in the APRS spec.

NMEA-183

This paper would not be complete if it didn't cover the NMEA-183 protocol. This is a serial communications protocol defined by the National Marine Electronics Association for the purpose of communicating between marine instruments. It should be noted that this protocol is not restricted to passing GPS data. It is important to the PIC-E-II project because almost all GPS instruments use this protocol to send out position information. A detailed description of the spec is beyond the scope of this paper. The NMEA specification is not available online, but there is a lot of information available on the web. Peter Bennett's website is the best place to find this information http://vancouver-webpages.com/peter/ . Interested readers should consult this page.

In brief, NMEA is a serial ASCII protocol that runs at 4800 baud. It is not RS-232 compatible, but the signal levels are close enough to work with most generic computer ports. Data is sent in what are called sentences. There are 46 standard sentences and a number of proprietary sentences. These sentences include not only position information, but also information about wind, water temperature, compass heading and various other information of interest to marine instruments. Unfortunately, there is no standard sentence that includes barometric pressure, humidity, or air temperature.

---

[1] MIC-E is a trademark of APRS Engineering LLC and Bob Bruninga WB4APR

These sentences vary in length, but are typically fewer than 100 bytes. They begin with a dollar-sign and end with an asterisk followed by a two-byte checksum. They are typically transmitted at a one-second interval. Most instruments cycle through a sequence of messages, taking several seconds to repeat the sequence. This means that data may be several seconds old before it is received.

An example GPS sentence is included below.

$GPRMC,212911,A,4915.607,N,12310.537,W,000.0,360.0,111198,020.3,E*61

PIC Processor

The microchip PIC processor midrange line offers a very low cost solution for simple embedded applications. The chip offers internal RAM and ROM. I/O devices such as line drivers and receivers, analog input, and serial I/O are included in the same package. For most applications, only an external crystal, three-terminal regulator, and rs-232 signal conditioner are needed to support the PIC chip. The chip comes in several configurations. Only chips with flash ROM are considered suitable for low-volume projects. This limits the selection of processor chips to the 16F84, 16F628, and 16F87x. The original PIC-E design used a 16F84 PIC processor. This chip does not include enough memory or I/O to support our requirements, nor does the 16F628. We chose the more powerful 16F876 chip. This chip has the added advantage of providing support for a low-cost in-circuit debugger.

The chip 16F876 architecture provides 8k words of program space, 368 bytes of RAM, and 256 bytes of EEPROM. One hardware UART is implemented in the PIC. A second serial interface is provided for synchronous serial communication. This port can be configured as a SPI bus interface. The SPI bus is a medium speed bi-directional serial bus that is used to connect a variety of low cost sensor and memory devices. The SPI bus uses three lines, data, clock, and chip select. Data is bi-directional.

It should be noted that a more advanced processor with flash memory is now available from microchip. This more powerful chip was not available when we started the design. A more powerful high-end chip which overcomes many of the midrange limitations is now available. We are evaluating the possibility of migrating to one of these chips in the near future. A high-end chip may make a multi-tasking real-time OS possible. This would significantly simplify the software design.

PIC Midrange Architecture Limitations

The midrange PIC processors lack a data stack. In theory, a stack could be implemented using part of the limited internal memory on the chip, but the overhead required to support an operating system context could not reasonably be supported without expanding the on-chip memory. Off-chip memory is either very slow or requires too many I/O pins. To compound these difficulties, available compilers for this chip do not make use of a stack. This prevents the use of a multi-tasking real-time operating system.

An 8-deep program address stack is implemented in hardware in the PIC. This limits program nesting to 8 levels. Only the top-most entry is visible to the software. This prevents the call stack from being saved during a context switch.

Although clock speeds for the PIC processors are advertised as high as 20mhz, it takes four clock cycles to execute one machine instruction. This reduces the effective speed of the chip to 5mhz.

The PIC architecture has only one data register. This means that it takes several instructions to execute even the simplest of operations. This restriction not only reduces the effective throughput of the processor, but also eats program space.

The PIC processor has only one level of interrupt. All other interrupts will be blocked while an interrupt service routine is executing. This means that ISRs must be kept short.

The lack of a data stack coupled with a lack of general purpose registers make it almost impossible to write re-entrant code. As a minimum, this implies duplicate copies of software functions that are used in the ISR context and the application context.

Both program space and data space are paged. Paging adds overhead to the system by requiring register settings before switching pages. This overhead reduces system performance. Page changes in program space are infrequent, but page changes to access on-chip RAM can be very frequent. Paging to access RAM can be minimized by positioning related data on the same page and by careful use of the few shadowed RAM locations.

The PIC processor has no hardware multiply or divide instructions. These operations are performed using a math library using software routines. Multiply operations and particularly divide operations should be avoided. Floating point operations in a time critical thread may have serious throughput implications and should also be avoided.

Development Tools

It is a team goal to provide a kit that is relatively easy to adapt. The choice of development tools is a key consideration. We were also aware that fellow amateurs are frugal. We attempted to strike a balance between cost and performance.

Development Environment

Microchip supplies a free development environment available for download from their website, the Microchip MPLAB IDE. This toolkit runs on Microsoft Windows platforms. It provides an assembler, debugger, simulator, and project facility. Note that it lacks a compiler. Without external hardware, the debugger only works with the MPLAB simulator.

Compiler

Assembly code is considered difficult for the average amateur to use, so we included a commercial compiler in our requirements. Several third-party compilers are available for the PIC chip. We restricted our selection to those that are low cost (less than $100). After evaluating several products, we selected the CCS PIC-C compiler. This is a C

compiler. The compiler has limitations, most of which can be overcome by the programmer. Our first copy had problems generating reliable code, but a later version fixed many of these bugs. The compiler does not conform to the ANSI standard, but the most recent version is close enough to make coding straightforward. I found that the compiler is more reliable if the C statements are kept simple. This also makes the code more readable for inexperienced programmers.

The CCS compiler comes with a support library that is specific to the PIC chip. These services provide drivers for most of the PIC devices, including the serial ports. They use blocking calls however. A blocking call locks out the processor until the service completes. As an example, a call to printf to print a character string does not return until the last character is queued in the output port. At 9600 baud, it takes over 1ms for each character to be transmitted. A string of 10 characters would take over 10ms to send out. During this period of time, the processor can service interrupts, but no other threads could run. Response requirements of the PIC-E-II design cannot be satisfied if blocking calls are used. I overcame this limitation by writing my own drivers.

Debugger

Various debuggers and in-circuit emulators are available. We restricted our selection to an inexpensive in-circuit debugger available through Microchip for less than $100. The debugger uses the Microchip MPLAB IDE software package.

The PIC-E-II design

Hardware Design

The schematic and assembly drawings for this board should be posted on the TAPR website in the near future. Those interested in examining the design should look there for the latest copy.

As stated earlier, the PIC-E-II hardware design started with the PIC-E design. It therefore resembles the earlier kit. Several changes were made to better support the new requirements. The earlier kit supported only one serial port, a GPS port. We added a second port to connect to a host computer.

The NMEA spec requires an opto-isolator to support the GPS interface. We wanted to make the sensor interface more generic than just GPS, so we supported the interface with an RS-232 compliant design. The signal level provided by the PIC is not in compliance with the RS-232 spec, so an external buffer/driver chip (MAX-232) was added.

The PIC processor includes one serial UART interface. A second serial port is needed for our implementation. The driver for this port is implemented in software. This means that each serial bit for this port must be shifted by the processor.

The inputs and outputs are conditioned using the same buffer/driver chip. Flow control for data from the host to the target is required to throttle packet data. This line is not properly conditioned as specified by the RS-232 spec. The MAX-232 chip only provides two buffered outputs which are used by the two serial port data output lines. A third was not available for this signal. Since the signal is a level rather than edge sensitive, the marginal signal level provided by the PIC output should work with most host processors.

The radio audio interface is supported with the same modem chip used in the original PIC design. Connection to the radio microphone was simplified to reduce parts count. Pin mapping from the TNC to the microphone can be done in the harness rather than on the board. It is necessary to toggle one of the mode bits when switching between transmit and receive. This is directly connected to the PIC TX line that goes through a transistor driver and then to the mic PTT.

There is not enough RAM provided in the PIC chip for packet buffering. We added an external RAM chip to the design. This chip is a Ramtron FM65640. It provides 8k bytes of buffer memory. Access to this chip is through the SPI serial bus running at one fourth the CPU clock speed. The slow access speed of the SPI interface precludes the use of this device for local data, but it is adequate for packet storage.

Software Design

The software implementation is too complicated to be described in detail in this paper. This section covers key parts of the software design. In some cases, code fragments from the software are included where it may help describe the design. Complete source for this project is available from the author on request. Code fragments included below are copyrighted. The author reserves all rights. Permission to use this code may be obtained by contacting the author.

Due to the physical limitations of the PIC processor, design and coding methods that are considered "normal good engineering practices" such as information hiding and encapsulation had to be ignored. During the implementation phase of development, code generation problems with an early version of the compiler forced me to write the code in an unusual style. This has resulted in an implementation that looks awkward when first examined. The code is well commented and should be easy to follow however.

I used a common coding standard for this code. Indents are four spaces. Close braces align with the corresponding statement. Macros are all capitol letters. Macros are used to make the code more readable or more portable. Macros represent a small block of code and are defined before they are used. An example of a macro used in the code is `LED1_ON`. This macro is defined in a header file that defines how the PIC chip is wired to the board. If the board layout changed, it would only be necessary to change this single definition to correct the code.

Real-time Considerations

Real-time requirements imposed by system require a multi-threaded solution. Each thread of execution supports a flow of data through the system. At a minimum, the following threads are required to implement the PIC-E-II design.

1. GPS receive – This device is implemented in software using a single I/O bit. Each received bit is sampled at the appropriate time until an entire byte is captured.

2. GPS transmit – This is implemented in software using a single I/O bit. Each transmit bit is loaded into the output port at the appropriate time.

3. Computer port receive – This interface is implemented in hardware using the PIC UART. An interrupt line is set whenever data is available from this port. Hardware flow control can be activated from this ISR to throttle receive data.

4. Computer port transmit - Computer port receive – This interface is implemented in hardware using the PIC UART. A hardware interrupt is available that indicates that the port is available, but the software does not enable this interrupt. The sending thread is responsible for polling the output port status.

5. APRS receive– This is implemented in software using a single I/O bit. Each received bit is sampled at the appropriate time until an entire byte is captured.

6. APRS transmit - This is implemented in software using a single I/O bit. Each transmit bit is loaded into the output port at the appropriate time.

Each thread has a real-time response requirement that is imposed by system requirements. Response times for these threads are listed below.

1. GPS receive – ½ bit time at 4800 baud = 104us. The receive data is sampled at the middle of the bit period. To insure accurate samples, this sample must occur before the end of the bit period.

2. GPS transmit – ½ bit at 4800 baud = 104us. It is assumed that the receiver implements an algorithm similar to the PIC receiver.

3. Computer port receive – 1 byte to set hardware flow control = 1.04ms. The receiver can buffer an entire received byte. When we detect the start of a transmission, we need to set the flow control bit before the end of the received byte.

4. Computer port transmit – no response requirement

5. APRS receive – ½ bit at 1200 baud = 416 us

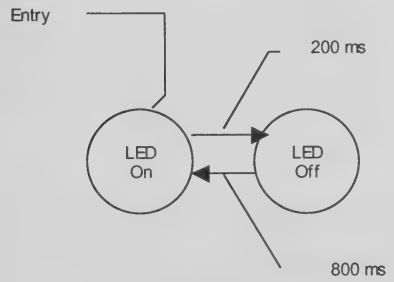
6. APRS transmit – ½ bit at 1200 baud = 416 us

The above response times are absolute worst case. A margin of safety is required for a robust system implementation. Reducing each of these times by 25% is advisable. Some of these time constraints are mutually exclusive. As an example, it is not possible to both transmit and receive on the APRS port at the same time, so the software does not need to consider satisfying both of these constraints at the same time.

The Grand Loop

As stated earlier, the PIC processor supports only one level of interrupt. Without multi-tasking, there are only two contexts that run in the processor, the ISR context and the application context.

Consider a simple thread that is only required to blink an LED once each second. A function such as this might be used to display a heartbeat. This is useful during the test and integration phase to indicate that the application software is cycling. The state diagram for this thread is shown below. When it first runs, it enters the LED on state where it turns on the LED. After 200 ms enters the LED Off state where it turns off the LED. After 800ms, it reenters the LED on state.

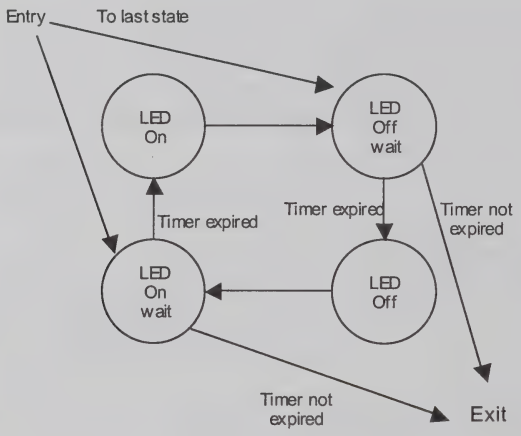


In a traditional embedded implementation, the above function would be implemented with a small task that executes two wait service calls. While waiting, the processor would be available to run other tasks. Without a multi-tasking real-time operating system, there is no scheduler that can load other tasks during the wait period. This means that during a wait condition, the processor is not available to run other ready tasks.

One method for overcoming the lack of an operating system is to design the software to operate in what is called a "grand loop". Since threads of execution cannot make use of multi-tasking OS, they are serviced in sequence within a single task context. Each thread is executed in turn within an infinite loop loop. Threads run until they gives up the

processor. Each thread is polled in sequence to allow it to continue execution. The polled thread is responsible to determine if it has the resources necessary to continue running. If not, it exits as quickly as possible. Under no circumstances may a thread block. When a thread begins execution, it must determine where it was when it last ran. It then must determine if it can continue execution. If resources required to continue execution are not available, the thread must exit.

Each thread is implemented as what is called a "state machine". Threads sequence through a set of "states" until they require a resource that is not available. Resources include data, devices, or events. An example data resource could be a message placed in a data queue by another thread. A device resource might be a free queue entry in the SPI interface. An event resource might be a timer event. In order to advance to the next state, the state machine must have the resources necessary to continue. It then completes as much of the thread of execution that it can until it requires an unavailable resource. The state machine remembers the state it was in when it stalled. It then exits, giving the next thread in sequence a chance to run. On the next invocation, the state machine checks to see if it has the resource it needs. Our example simple thread above looks like this when implemented as a state machine.



The thread above initially enters the 'LED on wait state'. It exits if the timer has not expired. When the timer expires, it advances the 'LED on state' where it turns on the LED, then advances to the 'LED off wait state'. If the timer has not expired, it exits.

Code for this thread is shown below. This thread uses a static variable ledHeartbeat_state to keep track of its machine state. It isn't apparent how the state machine advances from the two wait states to the LED_ON or LED_OFF state. The timer ISR is responsible for decrementing the ledHeartbeat_count. When this count reaches zero, it advances the ledHeartbeat_state.

```
#define LED_ON_PERIOD 20    /* 20 * 10 MS */
#define LED_OFF_PERIOD (100 - LED_ON_PERIOD)
                            /* 100 * 10 MS rate */
#define LED_WAIT_ON 0
#define LED_ON 1
#define LED_WAIT_OFF 2
#define LED_OFF 3
void ledHeartbeat(void) {
    switch ( ledHeartbeat_state ) {
        case LED_WAIT_ON :
        case LED_WAIT_OFF :
            break;
        case LED_OFF :
            LED1_OFF;
            ledHeartbeat_state = LED_WAIT_ON;
            ledHeartbeat_count = LED_OFF_PERIOD;
            break;
        case LED_ON :
            LED1_ON;
            ledHeartbeat_state = LED_WAIT_OFF;
            ledHeartbeat_count = LED_ON_PERIOD;
            break;
    }
    return;
}
```

All the threads in this application are implemented in a similar fashion. Time critical operations are done in the ISR context. The application thread is then advanced by the signaling thread or ISR using a method similar to this simple example. It might seem obvious to just set and clear the LED in the ISR. The above function could certainly be done in this fashion without impacting interrupt response time. A more complicated function would better be done outside the time-critical path however.

As described above, the main routine is an infinite loop. Each thread is executed in turn. The application main routine is implemented as shown below.

```
int main(void) {
    /* do initialization stuff …*/

    while ( 1 ) {
        /* Check for serial data to transmit
           via the computer port */
        serXmtSvc();

        ledHeartbeat();

       switch ( pice_state ) {
            case SYS_TEST :
                extmemtst();
                break;

            case SYS_APRS :     /* APRS mode */
                gps_rx();       /* check GPS data */
                aprs_time();    /* check to see if
                                   APRS message should
                                   be sent */
                aprs_xmit();    /* transmit queued APRS
                                   messages */
                aprs_rcv();     /* check for received
                                    messages */
            /* fall through and service KISS engine */
            case SYS_KISS :
                kiss_xmt_service();
                break;

        }

    }
}
```

Interrupt Service Routines

Some of the real-time response requirements are relative to external events while others are related to system time. Once we begin sending a byte of data to the GSP through the sensor port, we must shift a new bit into the output port within the specified period of time relative to our own internal clock. This is true for the AX.25 transmission as well. These service routines are managed as part of the system timer, running at 208us (4800 baud).

Received data from the GPS through the sensor port is being sent relative to the system clock of the GPS device, not our own clock. This means that we must capture and edge

interrupt at the start of a GPS data byte, and then set a timer to sample the eight data bits in that transmission. Capturing the edge and servicing this second timer require an interrupt service routine. The last interrupt service routine is the computer port receiver. This port is implemented by the PIC with a hardware UART, but the flow control bit has to be set before the receiver register overruns. An ISR catches the receiver in operation and sets this flow control bit.

It should be noted that the CCS compiler provides automatic registration of ISRs for devices supported by the CCS support library. The programmer normally doesn't have to deal with interrupt handlers unless he has special requirements. Event at that, the CCS compiler generates the code necessary to save and restore scratch registers and vector to the registered routines. I bypassed this code for performance reasons. I also added support code for saving and restoring registers. I did not show this support code below in order to make the code more readable.

```
INTERRUPT_ENTRY(pic_int) () {
    /* schedule timed events.
       Sensor port serial transmitter serviced here
       AX.25 transmit port serviced here
       AX.25 receive port polled here */
    sys_timer_isr();

    /* check the serial computer port to see
       if there is data */
    /* if there is, notify any pending recipient
       and set flow control */
    sensor_rcv_isr();

    /* check the serial computer port to see
       if there is data */
    /* if there is, notify any pending recipient
       and set flow control */
    ser_rcv_isr();
}
```

System Timer ISR

As stated above, the system timer runs at 4800 baud. Other timed events are derived from this rate as shown below.

```
void sys_timer_isr(void) {
    /* while timer interrupt */
    if ( T0IF != 0) {
        /* service the ax25 receiver port at 4 times
           the baud rate */
        ax25_rcv_isr();

        /* if 1200 baud event has expired */
        msCounter = msCounter - 1;
        if (( msCounter & 3)==0) {
            ax25_xmt_isr();/* service AX25 transmitter */
        }

        sensor_xmt_isr();/* service the sensor port */

        /* if 10 ms has expired */
        if (msCounter == 0) {
            /* reset the timer counter */
            msCounter = SYS_TICKS_10MS;

            /* check aprs transmit event */
            if ( aprs_timer_count > 0) {
                aprs_timer_count = aprs_timer_count - 1;
                if (aprs_timer_count == 0){
                    aprs_timer_state =
                     aprs_timer_state + 1;
                }
            }
            /* check LED heartbeat */
            if ( ledHeartbeat_count != 0) {
                ledHeartbeat_count =
                 ledHeartbeat_count - 1;
                if (ledHeartbeat_count == 0){
                    ledHeartbeat_state =
                     ledHeartbeat_state + 1;
                }
            }
        }
    }  /* if timer interrupt */
}
```

AX.25 receiver

Other implementations that I've seen for this device set a timer to go off at the center of each bit. The input bit is then sampled. Since the system timer for this design runs at 4800 baud, I chose instead to sample the input bit at each interrupt. When I detect a change in state, I count how many bit periods have expired. The system timer runs at exactly four times the AX.25 data rate, so converting to 1200 baud periods simply requires shifting the count over by two places. It is first rounded up by half a bit period.

At each state change, the number of bits received is computed. The code that does this processing is shown below.

```
/* count number of bit times (rounded) */
ax25_rcv_num_bits =
 (ax25rcv_period_ctr + 2)/4;
ax25rcv_period_ctr = 0;

/* if nobody wants the data */
if ( pax25_rx_state_callback == 0) {
    /* don't process it */
    ax25_rxisr_state = AX25_RXISR_IDLE;
    abort_detected = TRUE;
    return;
}

/* check for FCS */
if ( ax25_rcv_num_bits == 7) {
    fcs_detected = TRUE;
    ax25_rxisr_state = AX25_RXISR_FCS;
    ax25_rx_data = 0;
    ax25_rx_bit_ctr = 8;
    /* let them know data is avail */
    isr_ptr_increment(pax25_rx_state_callback);
    pax25_rx_state_callback = 0;
    /* they must request again */
    return;
}

/* check for ABORT */
else if ( ax25_rcv_num_bits > 7) {
    abort_detected = TRUE;
    ax25_rxisr_state = AX25_RXISR_IDLE;
    ax25_rx_data = 0;
    ax25_rx_bit_ctr = 8;
    /* let them know data is avail */
    isr_ptr_increment(pax25_rx_state_callback);
    pax25_rx_state_callback = 0;
```

```
        /* they must request again */
        return;
    }


for ( ax25_rcv_i=0; ax25_rcv_i < ax25_rcv_num_bits;
      ax25_rcv_i++) {
        /* we have a zero optionally followed by
           a series of ones.
           This isn't initially obvious.  A zero
           is indicated by a change in frequency.
           A one in indicated by no change.
           An interrupt occurs at each change of
           frequency (zero) its period is a minimum
           of one bit length.  It will extend by one
           bit length for each successive 1.
        */

        /* Zeros get skipped if they are stuffed
           or if it is the trailing zero at the end
           of an fcs */

        /* if we should skip this bit */
        if ( ( ax25_rxisr_state == AX25_RXISR_FCS) ||
           (ax25_rxisr_state == AX25_RXISR_STUFFED)) {
            ax25_rxisr_state = AX25_RXISR_RCV;
            continue;
        }
        /* otherwise shift in data  */
        ax25_rx_data = ax25_rx_data >> 1;
        ax25_rx_bit_ctr = ax25_rx_bit_ctr - 1;
        /* first bit is zero, rest are one's */
        if (ax25_rcv_i != 0) {
            ax25_rx_data |= 0x80;
        }
        /* if we received a character
           and someone wants it */
        if ( ax25_rx_bit_ctr == 0 ) {
            /* let them know data is avail */
            isr_ptr_increment(pax25_rx_state_callback);
            pax25_rx_state_callback = 0;
            /* they must request again */
            *pax25_rx_data = ax25_rx_data;
            /* they get the data */
            ax25_rx_data = 0;
            ax25_rx_bit_ctr = 8;
        }
```

```
    }
    /* AX.25 restricts no more than 5 ones in a row.
       When 5 are in the data steam, a zero is stuffed.
       We need to toss it on the next interrupt */
    if ( ax25_rcv_num_bits == 6) {
       ax25_rxisr_state = AX25_RXISR_STUFFED;
    }
```

Sensor Receive ISR (GPS port)

The GPS is connected to the sensor port. This port is implemented in software. The edge of the first data bit generates an edge interrupt. It should be noted that this interrupt is asynchronous to the system 4800 hz timer interrupt. From then on, a second timer (AUX timer) is used to time data bit samples. The thread state machine cycles through states as each bit is received until it counts the stop bit, at which time it re-enables the edge interrupt, stops the AUX clock, and gives the completed byte of data to a waiting thread.

```
void sensor_rcv_isr(){

    portb_val = PORTB;
    SENS_RX_CLR;

    /* if waiting for start bit, we need a logic one */
    if ( sensor_rx_state == SENS_SYNC) {
        if ( SENS_RX_LOW )
            return;
        sensor_rx_state = SENS_START;
        /* set timer interrupt for next bit time */
        AUXTIMERGO;
        /* no more edge interrupts until the byte
           is received */
        EDGE_INTERRUPT_DIS;
        return;
    }

    /* if waiting for a data or stop bit, we need
       a timer interrupt */
    if (TMR2IF == 0) {
        return;
    }

    sensor_rx_state = sensor_rx_state + 1;

    /* if we received a whole character
       and someone wants it */
    if ( sensor_rx_state == SENS_STOP ) {
        if ( psensor_rx_state_callback != 0) {
```

```
            /* let them know data is avail */
            isr_ptr_increment(psensor_rx_state_callback);
            /* they must request again */
            psensor_rx_state_callback = 0;
        }
        /* disable the auxiliary timer interrupt */
        AUXTIMERSTOP;
        /* next time we need a zero to one transition */
        /* reenable edge interrupts */
        EDGE_INTERRUPT_EN;
        /* wait for next sync */
        sensor_rx_state = SENS_SYNC;
        return;
    }

    /* shift in data  */
    sensor_rx_data = (sensor_rx_data >> 1);
    /* set timer interrupt for next bit time */
    AUXTIMERCONTINUE;
    if (SENS_RX_LOW) {
        sensor_rx_data = sensor_rx_data + 0X80;
    }
    return ;
}
```

Real-time Threads

Message Queues

The AX.25 message frame can be built from fragments in various parts of system memory. Some parts such as station callsign can come from EEPROM, other parts such as GSP position can come from internal RAM or external RAM. Rather than collect these parts and copy them into a single contiguous block of memory, I chose to implement a queueing mechanism that includes pointers to various parts of memory. This avoids the overhead of having to copy fragments into a contiguous block. Rather than queue this block, a pointer to each fragment is queued.

Each message queue entry is a 16-bit entry, consisting of a 3-bit type indicator followed by a 13-bit data field consisting of a pointer or literal field. The following queue entries are supported:

1. Text Data- Data field points to a null-terminated string in program memory.
2. ExtRAM – Data field points to a null-terminated string in external memory.
3. RAM - Data field points to a null-terminated string in internal memory.
4. EEPROM - Data field points to a null-terminated string in EEPROM memory.
5. Literal – Data field has a single byte to transmit.
6. Begin CRC – Don't transmit anything. Initialize the global CRC value.

7. Transmit CRC – Data filed contains nothing. Transmit the two-byte global CRC value.
8. Transmit KISS packet – Data field points to an FESC terminated KISS packet in external memory.

Two queues are used in this software design, the AX.25 transmit queue and the computer port transmit queue. Using the above queue entries, the process of sending a mic-e encoded packet is coded as shown in the following example. Each statement queues an entry in the AX.25 transmit queue.

```
TRANSMITCRCBEGIN;
transmitRamString(mice_buf_dest);
transmitEeString(CALLSIGN);
transmitEeString(PACKET_PATH);
transmitLiteral(AX25_CONTROL_FLD_UI);
transmitLiteral(AX25_PROTOCOL_ID);
transmitRamString(mice_buf_info);
transmitEeString(STATUS_TXT);
transmitEeString(COMMENT);
TRANSMITCRC;
```

Time Keeping for APRS messages

Various timed events need to be managed in order to comply with the APRS and AX.25 specs. These events include transmitting position information at a periodic rate and managing the packet transmission itself. This includes transmit hold-off, persistence, and time from key closure to first data byte. A timer thread is dedicated to managing these events. All of these timed events are multiples of 10ms. The system timer ISR keeps track of 10ms events. When a programmable number of periods have elapsed, the state of this thread is advanced.

GPS Receiver Thread

Data bytes from the sensor port are collected by the sensor receive ISR. When a completed byte is available, it is forwarded to the GPS receiver thread. This thread processes the Recommend Minimum Specific GPS/TRANSIT Data sentence (RMC). This sentence is in a fixed format. The thread recognizes the sentence header and processes the data by counting byte position. As each field is received, it is converted to mic-e format and saved for later transmission as part of the station position packet.

APRS Receiver Thread

Data bytes collected by the AX.25 receive ISR are stored in external memory by this thread. As they are received, a running CRC is computed. If the CRC indicates a valid packet, a check is done to see if an external host is connected. If an external host is attached, the record is closed and forwarded to the computer port queue for transmission as a KISS packet to that host. If a host is not connected, a check is done on the packet to

see if it is addressed to this station. If it is, the record is closed. If external memory has space for an additional packet, a new record is opened.

KISS Transmitter Thread

Any time that a host is connected to the TNC it can send a packet. This packet is queued in external memory.

Project Status

Currently, the prototype hardware is in test. A version of the software is in integration. It supports all the requirements defined in the software functional spec in the appendix. It transmits and receives packet data, receives and parses GPS sentences, logs packets, forwards packets to a host computer, and performs self-test. Significantly more test is needed before it can be distributed to the amateur community.

More work is needed in the APRS receive and KISS transmit threads. Both of these threads require a host application to run before they can be debugged.

Program space has become an issue on the target. I am at over 90% utilization. I've gone through a couple of redesign efforts to reduce code size. One of these efforts reduced code size by removing any information hiding in the design. The implementation can be scaled. As an example, self-test can be disabled by using a pre-processor directive. I used 16-bit data pointers when I started the design. This caused code generation to produce a lot of code to manage paging when accessing data. I suspect that reverting to 8-bit pointers and making use of special CCS services to access pages 2 and 3 for message queues would reduce the code size by as much as 20%.

Where to go from here

Work on this project began more than 18 months ago. It has been a challenge finding time to complete the project. Considerable work has been done on the software, while work on the production board layout and fabrication has languished.

Several attempts have been made to get help with board layout, packaging, testing, and host software application. Enthusiastic volunteers have come forward, but there is no follow-through. We all have day-jobs and family responsibilities that reduce our free time. It is easy to volunteer for a project such as this, and then lose interest when the size of the job becomes apparent.

Work on host application has also stalled. The team member who signed up to do this assignment has vanished. This software allows the TNC to connect to a host computer and upload stored messages. It also allows downloading setup parameters and sending asynchronous packets. I'm hoping that the winter months might provide more time from the rest of the team.

Reference List

- Internetworking with TCP/IP, Douglas E. Comer
- AX.25 Link Access Protocol for Amateur Packet Radio

Appendix-1 System Functional Spec

The product will be user programmable. It will either include all necessary support hardware to perform in-circuit programming, or will be programmable using an external device. If external hardware is required, it will cost less than $25.

Source for the product will be available for modification and upgrades. Source will be built using development tools that cost no more than $100. The availability of a C compiler is highly desirable, but not a requirement.

The product will run on 12v(nominal) external power, 1 amp max. An internal replaceable backup battery may be needed. The product will be fuse protected.

The product will provide a bi-directional interface to an external radio at 1200 baud using AX.25 audio tones. It will provide the ability for the software to detect a busy channel.

A method will be provided to sense the microphone PTT line. The product will be capable of detecting key closure of the microphone.

The product will be capable of servicing all interface ports simultaneously. Full duplex operation will be supported.

The product will display three indicators for power, PTT, and channel busy.

The product will accept data from an external device such as a GPS or weather station. Data will be received serially using the RS232 protocol at baud rates between 1200 and 4800 baud. The product will be capable of transmitting to the same device at the same baud rates.

Support will be provided for the Tripmate GPS. This support includes jumpers to supply DTR and to wrap tx to rx. A jumper will be provided to supply 5v power or 12v power to pin-9.

The product will interface with an external serial terminal device using a bi-directional RS-232 port running at 9600baud. Hardware flow control (DTR only) will be provided to throttle the external device. This device will be a terminal, printer, or computer device.

The product will have the capacity to save non-volatile parameters. This storage area will be sufficient to save call-sign, path, station icon, position data (for fixed station), position comment, status text, transmit method (key, periodic), and transmit interval. A minimum of eight bytes of application specific parameters will be also available.

A method will be identified to set time and non-volatile parameters using the terminal port.

The product will have the capacity to internally record a minimum of four APRS formatted messages for later retrieval.

A method will be identified to display recorded messages using the terminal port.

Appendix-2 Software Functional Spec

Overview

This document defines the software functional requirements for the PIC-E-II APRS TNC.

The PIC-E-II will function as a stand-alone APRS station. It will operate in the following basis modes:

1. Startup – in this mode of operation, the PIC-E-II will print out a startup banner including software version. It will then enter the system default mode of operation (see configuration parameters).
2. Idle – in this mode of operation, the PIC-E-II will monitor the terminal port for user commands.
3. Self-test – in this mode of operation, the PIC-E-II will test external memory and peripheral interfaces.
4. KISS – in this mode of operation, the PIC-E-II will send and receive packets to a host system using the KISS protocol.
5. APRS – in this mode of operation, the PIC-E-II will transmit position reports at a programmable rate. It will receive and log messages that are addressed for the station.
6. Program – in this mode of operation, the PIC-E-II will accept and load a new image.
7. Update – in this mode of operation, the PIC-E-II will accept and load configuration parameters.
8. Monitor – in this mode of operation, the PIC-E-II will provide a user interface to allow examining RAM, ROM, and external RAM.
9. NV-Load – in this mode of operation, the PIC-E-II will accept non-volatile parameters.

Note: Due to program memory size constraints, it may not be possible to support all these modes in one load image. At this point, it looks possible.

Idle

During idle, the PIC-E-II will monitor the terminal for a user command. User commands will consist of one-byte codes defined as follows:

1. T – Test
2. K – KISS
3. A – APRS
4. N – Parameter Update

5. M – Monitor
6. E - Echo

When a command is recognized, the system will exit idle and enter the commanded mode. If a command is not recognized, a warning message will be displayed and the system will remain in idle. An example warning message is shown below.

??? [TKANME]

Self Test

Self-test mode will load a test pattern into external memory and verify contents. It will forward characters received on the terminal port to the sensor port, implementing hardware flow control to prevent data loss. It will send a test pattern to the modem port to allow level adjustment. Self-test mode will terminate when an escape character is received on the terminal port. When received, the system will enter the idle mode.

KISS

During Kiss mode, the PIC-E-II will read packets detected on the modem port. It will compute a CRC on the received packet and compare it with the packet FCS. If the CRCs do not match, the packet will be discarded. Otherwise, it will examine the destination address for a match with station ID. If a match is found, the packet will be formatted as a KISS packet and forwarded to the host system via the terminal port. If the packet is not addressed for this station, the first entry in the path will be examined for a match with the station alias. If a match is detected, the packet will be reformatted and retransmitted. Packets received from the host will be transmitted through the modem port. Kiss mode will terminate when directed by a return command packet from the host or when an escape character is received on the terminal port outside a packet. When received, the system will enter the idle mode.

In addition to data packets, the following command packets will be recognized and processed by the system.

1. TX delay
2. Persistence
3. Slot time
4. TX tail
5. Full Duplex
6. Set Hardware
7. Return

Note: Storage will be provided for buffers with a maximum size of 1024 bytes. This will support KISS packets of a length of 1022 bytes max. Two transmit and two received buffers are planned. Hardware follow control is supported to control data flow from the host to the target. Support is not provided for flow control from the target to the host.

APRS

While in APRS mode of operation, the system will operate as either a mobile or fixed station. As a mobile station, it will receive position reports from an external NMEA-183

compliant GPS sensor at 4800 baud. As a fixed station, it will use a fixed position defined in system configuration parameters. It will format and send a position report at a programmable interval. This message will use the mic-e format. It will read packets detected on the modem port. It will compute a CRC on the received packet and compare it with the packet FCS. If the CRCs do not match or if the packet is too short, the packet will be discarded. Otherwise, it will examine the destination address for a match with station ID. If a match is found, the packet will be logged in memory for later retrieval by the host system. An external beeper will briefly sound to indicate data reception. A front-panel lamp will blink to indicate the number or stored messages. A notification will be sent to the host system (optionally connected) in the form of a single byte code byte (0x11). An auto-reply message will be sent. If the packet is not addressed for this station and the beacon mode is enabled, the first entry in the path will be examined for a match with the station alias. If a match is detected, the packet will be reformatted and retransmitted. When a query request is received from the terminal port, stored packets will be forwarded to the host system using the KISS protocol. A query request will consist of a single byte (0x11). When a query request is received, the system will send the oldest stored packet to the host and then delete it. If additional packets are stored, the packet will be followed by another ready byte code (0x11). If no packets are stored, an empty byte code will be returned (0X13). Up to twelve (tbr) packets will be stored, each with a maximum length of 256 bytes. When used as a mobile station, the system position received by the GPS can be saved as a configuration parameter. When an 'S' character is received on the serial port in mobile operation, the current position will be saved. The current position can be queried by the host using a ^E (0x05) character. When received on the terminal port, a copy of the last GPS RMC sentence will be sent to the host. During APRS operation packets can be sent from the host using the KISS message format. No formatting or validation will be performed by the TNC. The message will be sent out as soon as a time slot is available.

APRS mode will terminate when an escape character is received on the terminal port. When received, the system will enter the idle mode.

(Future enhancements) Various sensors other than a simple GPS could be connected to the PIC-E-II, including integrated instruments (NMEA instruments that include weather data) barometric pressure sensors and weather stations.

(Future enhancement) Sending the position of received packets out to the GPS using WPL NMEA sentence.

(Future enhancement) Sending received NMEA payload data out the terminal port for remote processing.

Note: It is possible to recover stored packets after re-powering. I may add this feature. The FRAM chip that we selected has 8Kbytes of non volatile storage. 4 K is assigned to KISS packet buffers and the remaining 4 K is assigned to APRS packet storage. These modes are mutually exclusive, so some savings could be achieved by combining these buffers.

Update

The Update mode of operation will allow system configuration parameters stored in non-volatile memory to be updated. During an update, the host system will send these parameters to the PIC-E-II using a hex record format. When the system enters this state, an 'N' will echo on the terminal port. Subsequent hex characters will not echo. The following configuration parameters will be stored. (Consult eeprom.h for data definitions.)

1. Callsign
2. Position Lat
3. Position Lon
4. Status Text
5. Group Code
6. Packet Path
7. Transmit Method
8. Position Method
9. Symbol Code
10. Symbol Table ID
11. Transmit Rate
12. My Alias
13. Mode (Mobile, Fixed)
14. TX delay
15. Persistence
16. Slot time
17. TX tail
18. Full Duplex
19. Default system mode (T,K,A,M)
20. Beep (Mine, All, None)
21. Beacon (On, Off)
22. Checksum

Update mode will terminate when an escape character is received on the terminal port. When received, the system will enter the idle mode.

(Future enhancement) A provision to upload configuration parameters to a host system might be useful.

Monitor

The monitor mode will allow the user to examine internal and external memory. The system will accept monitor commands from the terminal input. Commands will consist of a single command character followed by a hex number terminated by 'CR'. Commands recognized are listed below.

1. Display RAM 'D'
2. Display EEPROM 'E'
3. Display External Memory 'X'

If a command is not recognized, a question mark will be displayed. Monitor mode will terminate when an escape character is received on the terminal port. When received, the system will enter the idle mode.

Echo

This mode will allow testing the sensor port. It is also useful to verify the proper connection of a GPS. Characters received on the console port will echo to the sensor port. Characters on the sensor port will echo on the console port.

Loader

Program mode will be entered when the system powers up with a program jumper in place. This code will reside on the last page of program memory, space normally assigned to the ICD. It will be built as a separate image. When entered, the system will display a startup banner including the version number of the loader program. It will accept a program file from the system terminal port using an S-record format. It will load this image into program space, protecting its own address space and the reset vector from being overwritten. Program mode will terminate when a load image is received. It will then boot the loaded image.

NB: Kits will be shipped with the loader image burned into flash.

Support Software

The following support software will be required for system operation.

APRS Host Software

This software will allow a host computer system to connect to the PIC-E-II to allow examining stored messages and to set configuration parameters. Received packets should be parsed and displayed in an intelligent fashion. A clear display should be provided to set configuration parameters. This software will be described in its own functional spec. Suggested host platforms include Windows, CE, Palm, and Unix variants.

Host Loader

This software will allow a user to re-flash the program image in the PIC-E-II. A simple terminal program such as Hyperterminal or tip might be sufficient to support this feature. Some sensible method to set configuration parameters is needed however. If this APRS host software is not provided in the initial release, the loader will need to support this. Gary's application needs a method to load configuration parameters too, and his software doesn't use the APRS host software. Some coordination is needed here.

APRSWXNET/CWOP - a beneficial partnership of NOAA, Amateur Radio, And Other Good Citizens

Russ Chadwick, KB0TVJ
NOAA Forecast Systems Laboratory
Boulder, Colorado

Abstract

APRSWXNET/CWOP is a group of amateur radio operators and other citizens who have an interest in weather measurements and contribute those data to NOAA for important uses. This paper describes how the data is collected via packet radio and via the Internet. Where the data go and how they are used is also covered. Maps of the location of the data contributors for both packet radio and for the Internet are presented.

Key Words

Public service, meteorological data, citizen contributors, weather prediction, weather stations, amateur radio, citizen weather, packet radio, Internet

Introduction

Originally, APRSWXNET was a way to collect amateur radio weather data through the findu.com server and send them to the National Oceanic and Atmospheric Administration (NOAA) for research use by Forecast Systems Laboratory (FSL). The success of this effort and the amount of data routinely collected have led to use by other research labs and by operational parts of NOAA.

Currently, there are nearly 600 amateur radio operators who have sent data to NOAA through this system. Most of these are from home weather stations with suitable APRS programs generating digital data that a terminal node controller (TNC) interfaces to a radio. The radio sends weather data packets out on a frequency of 144.390 MHz and these data are repeated by APRS digipeaters and often received by APRS Internet Gateways. The packets are checked and then sent to the APRS IS (APRS Internet System). The findu.com server monitors the APRS Internet data stream and saves the data in a database. Every 15 minutes, the server assembles a file where each line represents the data from the last data packet from a particular station in that 15-minute period. The NOAA FSL Central Computer Facility acquires that file by anonymous FTP every 15 minutes. This arrangement has worked well.

Expansion to Citizen Weather

It became clear that the system and method of weather data collection from widely separated citizen weather stations was also applicable to persons not holding an amateur radio license if there was no radio transmission of the data into the APRS IS. If the user has internet capability (dedicated or dial-up) and can send properly formatted APRS weather data packets to any APRS server, those packets will get to the findu.com server and be available for routing to NOAA.

The available APRS programs interface with only a limited number of home weather stations. However, the Weather-Display program, written by Brian Hamilton of New Zealand, readily interfaces to almost all of the home weather stations currently available. In addition, Weather-Display was already finding use as a weather application under APRS programs. All that remained was to add the capability to connect to an APRS IS server as an unverified user, send a properly formatted APRS weather data packet, and then disconnect. This way of getting data to findu.com and on to NOAA can be used by both hams and non-hams alike.

The addition of citizen weather capability brought about a change in the name of the program. The original name of APRSWXNET was expanded to APRSWXNET/Citizen Weather Observer Program and this was shortened to APRSWXNET/CWOP. Many people have shortened this even further to CWOP. The program logo, developed by Dave Helms (CW0351), is shown in Figure 1.

Figure 1. Logo for APRSWXNET/Citizens Weather Observer Program.

Currently, (Aug 1, 2002) there are about 520 non-ham citizens who have expressed an interest and joined the group. The total membership is about 1100. However, these numbers are growing steadily as more citizens and amateur radio operators become aware of the activity. Membership lists and links to each of these members and their data along with much other useful information are given here,

http://wxqa.com http://dhelms.mystarband.net/cwop.html

MADIS

The APRSWXNET/CWOP weather data contributed to NOAA goes to the Meteorological Assimilation Data Ingest System (MADIS), developed and operated by FSL. Information about MADIS is given on this web page,

http://www-sdd.fsl.noaa.gov/MADIS/index.html

Data of various types, including surface meteorological data, radiosonde data, profiler data, hydrologic data and automated aircraft data are acquired, quality checked and managed. The purpose is to add value and make them useful for data assimilation, numerical weather prediction (NWP) and other hydro meteorological applications.

The data collected through MADIS is subjected to hourly quality checks that indicate if a station produces data of questionable quality. This quality checking becomes very important when the primary use of the data is as input to NWP programs used for automated forecasting. Any ham or non-ham that contributes data to APRSWXNET/CWOP can view the quality checking results on an hourly, daily, weekly or monthly basis through this FSL web page,

http://www-sdd.fsl.noaa.gov/MSAS/qcms_messages.html

The main purpose here is to provide objective feedback to the station operator to indicate if a problem exists so that corrective or suitable maintenance actions can be taken.

Organizations can subscribe to receive parts or all of the MADIS data set, which is available via ftp or by using Local Data Manager (LDM) software available to university users. The access to the database has also been designed so that the data formats are entirely compatible with the NWS Advanced Weather Interactive Processing System (AWIPS). This means that the MADIS data set, and in particular the APRSWXNET/CWOP mesonet data are easily used, processed and displayed by the standard work station deployed at every NWS Weather Forecast Office.

Users of APRSWXNET/CWOP Data

The original goal was for the data to become part of the research data stream in FSL. Not only was this goal achieved, but also it was soon passed. The APRSWXNET/CWOP data were used with other mesonet data in high-resolution, short-term NWP research in FSL as well as in the National Center for Atmospheric Research (NCAR) supported by the National Science Foundation. In addition to these users, the mesonet data now go to the National Center for Environmental Prediction (NCEP) and to a number of NWS Weather Forecast Offices (WFO). A detailed explanation of how the APRSWXNET/CWOP data are used at the MelbourneWFO is presented here,

http://www.srh.noaa.gov/mlb/ADASLDIS.html

There are also other uses of the data carried out in a less structured way. For example, the Long Island Railroad uses the data to monitor the weather conditions along their routes. This is especially useful during the winter months. Also, the Lake Tahoe Fire Protection District uses the data for wildfire assessment. This is especially useful during the summer months. The point being that there is a wide range of uses for this data.

Identifying APRSWXNET/CWOP Stations

Member stations have three different means of identification in the APRSWXNET/CWOP database. First is the provider ID, which is a 5 or 6 character name starting with ap or CW (like apxxx or CWxxxx). The x indicates a number 0-9. A provider ID like apxxx indicates that the data is from a ham. A provider ID like CWxxxx can be for either a ham or a non-ham. The second means of identification is the NWS ID, which is a 5-character name like APxxx or Cxxxx. This identifies the station on the NOAA mesonet display and also identifies the quality checking results for each station. The third means of identification is call sign, which for hams is the call sign or alias that their data are carried under on the findu.com server. The call sign ID can also be CWxxxx for non-hams or for hams wishing to use it and is assigned when that person completes a sign-in process on findu.com.

The process of registering a station in APRSWXNET/CWOP starts with getting a "call sign". For hams that want to be registered under their call, they already have this "call sign", i.e. their ham call with or without a SSID attached. Others simply need to fill out an Internet form on findu.com and they will be assigned a CWxxxx "call sign" where the xxxx value is sequentially assigned. The registration process is to simply check on a findu.com map and verify that the plotted location is correct and send e-mail to chadwick@fsl.noaa.gov indicating that the location is correct. Then that call sign will be added to the list of stations which have their data transferred from the findu.com server to the FSL Central Facility every 15 minutes. The reason for this registration process is to ensure that the location for the weather data is correct. The station operator is the best person to verify the location of the station.

This registration process gives rise to three distinct classes of members. The first is of registered amateur radio operators who have sent data. The second is of citizen weather operators who have filled out a web-based form, sent in data, verified their location and registered their station. The third is of citizen weather people who have filled out the web form, may have sent data, but have not verified their location or registered their station. Maps for these three classes of members are shown in Figures 2, 3, and 4 below.

Acknowledgements

Many unselfish people have contributed in no small way to the ongoing success of this unique cooperative program. A list of those people would include Steve Dimse K4HG; Patty Miller FSL4; Mike Barth FSL4; Dave Helms NWSH; Dick Stanich KB7ZVA; Bill Diaz KC9XG; Brian Hamilton W-D; Joe Schmidt W4NKJ; Scott Stierle FSL3 and Joe Chadwick KB0TVK.

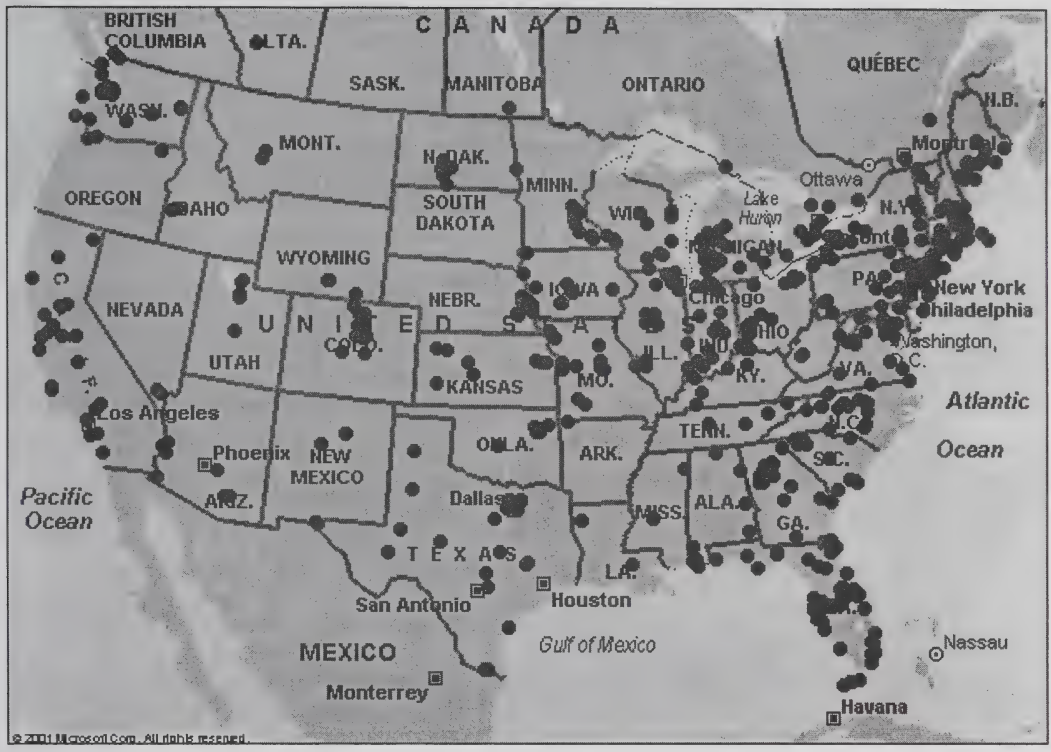


Figure 2. There have been nearly 600 amateur radio weather stations registered into the APRSWXNET/CWOP database. This map shows the locations of those near the contiguous United States. Each of these stations has or is contributing weather data to APRSWXNET/CWOP through the findu.com server and those data are being routed to NOAA.

To register an individual station, there must be reasonable weather data in the findu.com database. It is assumed that if an amateur radio operator has sent weather data to findu.com for more than a few days, then the plotted location must be correct. This map was created 01 Aug 2002.

Figure 3. There have been over 600 persons (ham and non-ham) who have filled out a form on findu.com indicating an interest in APRSWXNET/CWOP. They have each been assigned a CWxxxx designator. Most of the hams decide to use their ham call sign rather than the CW designator.

This map shows the locations of the approximately 240 persons in the contiguous United States who chosen to use the CW designator, and have sent data to findu.com under that designator, then verified the location plotted on findu.com and have registered their stations with NOAA. This registration procedure (after they have sent in data under the assigned provider CW designator) safeguards against data being assigned to the wrong location. This map was made on 01 Aug 2002.

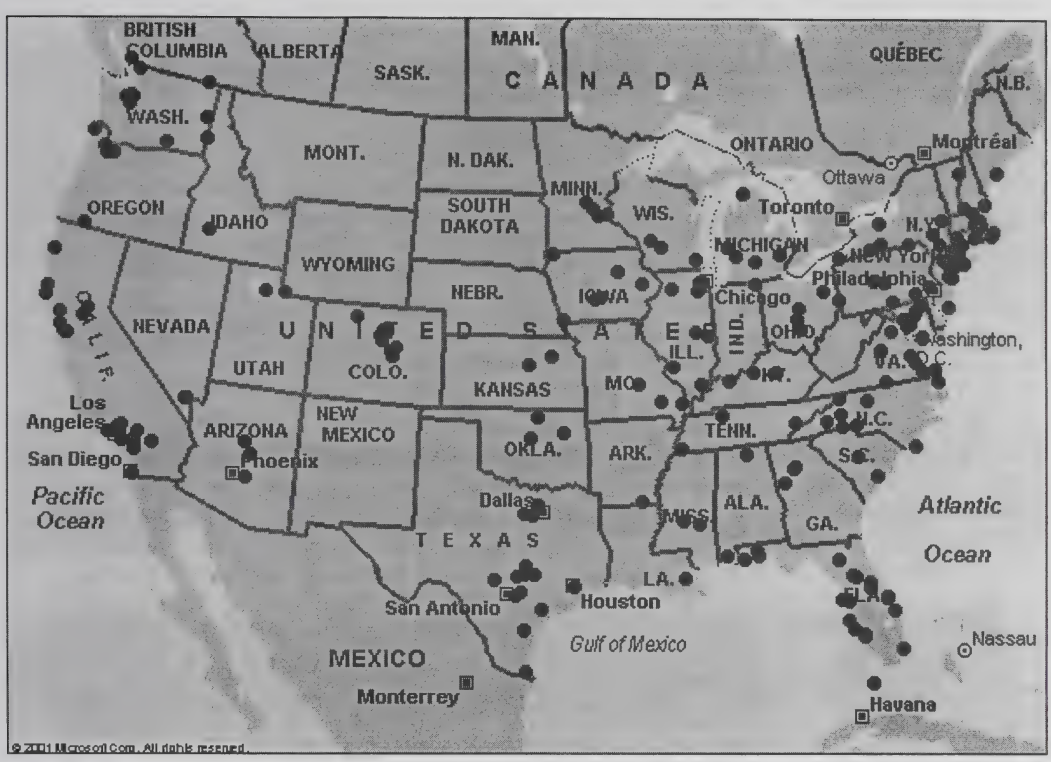


Figure 4. Of the over 600 persons (hams and non-hams) who have filled out a findu.com form, some are not heard from again. If e-mail to them bounces, they are removed from the APRSWXNET/CWOP database. Of the rest, some send in data and some don't.

This map shows the given locations of about 280 persons in the contiguous Unites States who have filled out the web-based form, possibly contributed data, but have not verified their location as given on findu.com. Some of these are operating stations contributing data, but the location associated with the data cannot be verified. None of the data from these stations are passed from findu.com to NOAA. These data are displayed on findu.com, but are not displayed on the NOAA mesonet web page and are not checked for quality. This map made 01 Aug 2002.

Automatic Packet Reporting System: Building a Large Scale Geospatial Database

James Jefferson Jarvis KB0THN
kb0thn@aprsworld.net

Abstract

The purpose of my research is collecting and analyzing position data from amateur radio's Automatic Packet Reporting System (APRS). APRS equipped vehicles transmit their latitude, longitude, course, and speed. This data is received via VHF radio and aggregated into the APRS Internet stream.

I designed and implemented a system that automatically collects and catalogs the entire APRS Internet stream into a relational database. Because APRS data is expressed in various formats, I wrote a parser that translates the data to a common format. My software collected more then 52,000,000 data points in 3 months.

I developed software to analyze position data. One program determines what polygon latitude / longitude points falls inside, and locates a station within a political boundary. For use in detecting roadway traffic problems, another program searches digital maps for the distance to the nearest road. Overall I wrote approximately 10,000 lines of computer source code. Currently I am investigating using neural networks to detect characterize roads and be able to detect anomalies, such as traffic jams.

Keywords: APRS, datamining, database, GIS

Introduction

Thousands of amateur radio operators drive around the United States each day transmitting their latitude and longitude to anyone listening. Tens of thousands of companies use Automatic Vehicle Locating technology to dispatch and track their fleets. Police, fire, and emergency medical service groups use Global Position System (GPS) tracking equipment for their day-to-day operations. A wealth of geospatial data is generated each day from these activities and many more. The purpose of my project is two-fold: to develop techniques and a system for effectively collecting this data, and to analyze the data to provide useful output products.

The first system, used by amateur radio operators, is called Automatic Packet Reporting System or APRS. APRS was developed in 1992 by Bob Bruniga as a means of tracking Naval Academy boats during summer cruises. Since then it has evolved to become a complex mesh of protocols for transmitting position, weather, messages, and status information by way of digital packet radio. In most cases APRS uses GPS for determining the user's precise latitude, longitude, and elevation anywhere on earth. The position is digitally encoded into a packet radio signal then transmitted by VHF radio. Other information can be included with each transmission such as course of movement, speed, messages, and text-based status information.

From 1992 until 1996 APRS was a collection of radio-based networks spread out across the country. For people to communicate with each other they had to both be within range of the same radio network. In 1996 Steve Dimse developed a means of combining all of the different networks into a common APRS network on the Internet. In each area with a radio-based network there would be an IGATE computer running special software to receive the messages from the radio and transfer them to Dimse's Internet

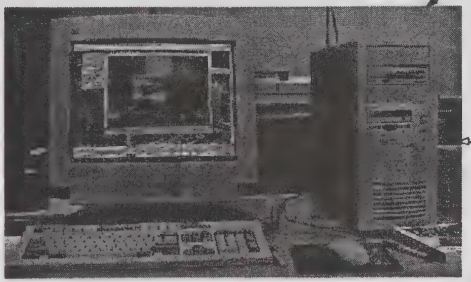
Position Reporting Stations

Worldwide
APRS Network

APRS
Internet Stream

APRS Internet
Gateways (IGATEs)

Data Collection



Database

server. Once the data was combined at the Internet server, anybody could connect to it and receive an aggregate stream of all of the information from all of the networks. When Dimse's system first came online he was aggregating data from San Francisco, Los Angeles, Nashville, Atlanta, Miami, and New Jersey. The data represented, at most, a few hundred users, accounting for a few thousand packets each day. Since 1996 APRS has grown in leaps and bounds. Amateur radio operators all around the world are building APRS networks. Today the APRS Internet stream aggregates over 600,000 packets per day from thousands of networks and users around the world.

Procedure

To facilitate analysis of the APRS data I designed several systems for storing and retrieving the data delivered by the APRS Internet stream. From the beginning of my research it was obvious to me that I would have to write a lot of custom software. I selected the Linux operating system for my research. Linux is a free, open source code, Unix-like operating system that works extremely well on a variety of hardware platforms. I avoided using proprietary software because I did not want to be locked into a particular software vendor that could not meet all of my needs. By using open source software I was free to learn from, and build upon, thousands of different programs. I chose to write most of my programs in the programming languages C and PHP and store data using MySQL relational database software.

APRS Database Design

I designed five C structures for use in the parser: a raw packet structure, a message structure, a position structure, a status structure, and a weather structure. The structures all closely resemble the table layout in the MySQL database. When the parsers are passed the raw packet information they are also passed a structure to fill with the extracted data. The individual parsing routines are responsible for converting the many different APRS formats into the common database structure. In many cases the parse routines call other functions help process the data.

After the packet is parsed and the appropriate structure is filled, the structure is inserted in the database. I designed the database to have each packet be keyed on a system-generated unique ID. Every packet is inserted into the raw table that contains the unparsed packet along with date and time, source, destination, and routing information. If the packet was successfully parsed into one of the five structures, then the unique packet ID and parsed values are inserted into the appropriate database table.

First Attempts

My first attempt at collect the whole APRS Internet stream was the spring of 2001. I wrote a program in C called net2db. Net2db would connect an APRS Internet server and store in the MySQL database each packet along with a timestamp of when my program received it. With net2db I was collecting over 3 packets per second and handing each one to the database. After about an hour the network connection would time out and the program would hang, so I had the system automatically restart net2db every 10 minutes. Although this system worked for collecting raw packets it had a number of problems. Besides text-indexing the whole packet field, there was no way to index the database, making it very computationally expensive to retrieve specific packets. Coupled with an inefficient database design and a five-year-old machine it was nearly impossible to retrieve data using this technique.

APRS Parser

To replace net2db, I wrote a full-fledged APRS parser that would not only store the raw APRS packets in a retrievable form, but it would also decompose packets into generic data structures. A parser is a program of function that breaks data down into specific elements. I named the parser message, because that was the first parsing function I wrote. APRS is relatively complicated protocol because it was never really designed. Instead the protocol was written around existing implementations.

I designed the parser to be tolerant in the way which it parses packets. I wanted it to make an attempt to parse the packet, but not to crash if it could not. This decision was not made lightly, but I eventually concluded that with a large dataset it would not be statistically significant to have erroneous information occasionally stored in the database.

After initializing everything and establishing connections to the Internet server and MySQL database the main function simply passes each received packet to the routine that begins the parsing. Every packet is run through a parser that separates the source, destination, routing information and the body of the message. The process function determines what parsing routine to hand the packet to based on the first information character in the body of the packet.

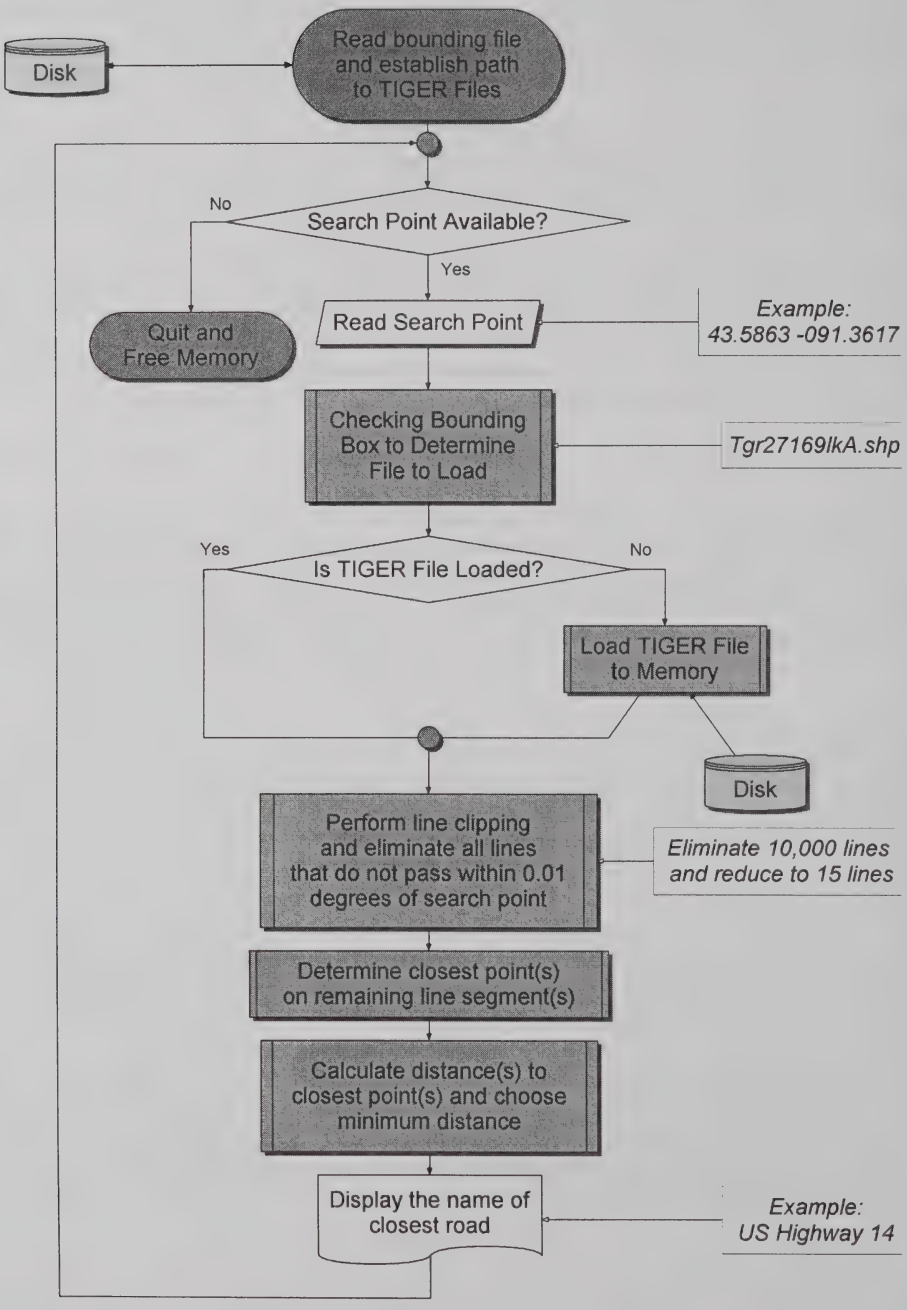
Locating the Nearest Road

Effectively modeling traffic flow has been a long-standing goal of the computer science community. I think that it may be possible, with enough diverse data, to actually monitor traffic for anomalies. The first step in recognizing traffic problems is to determine what roads vehicles are on. I wrote a program called shpdump-arc, which takes a latitude / longitude pair as input and provides the distance to, and name of, the closest road. I used US Census Bureau TIGER maps as my roadway base maps. The TIGER maps define roads as a series of line segments. The TIGER maps come in 7 gigabytes of map files.

Geometrically, the closest distance between a point and a line will be an imaginary line running perpendicular to the original line and intersecting the search point. By adapting this general principle to work with line segments, it is possible to calculate the distance from a search point to every other line segment on the map. Since the line segments represent roads, the line segment associated with the smallest calculated distance will correspond with the nearest road. If the point is not within a few meters of a road then the point cannot be considered as being on the road. Therefore only roads near the search point need be considered. I used a line clipping algorithm to decrease the lines segments searched. Line clipping is the process of determining what line segments fall within a search area and then searching only those segments. Determining what road is closest to the search point is simply finding the line-segment with the smallest distance to the search point.

Determining what road a vehicle is a major step in identifying congestion and other traffic problems. Neural networks could be used determine whether or not the current vector of the vehicle is consistent with the characteristics of previously examined data about the road. In the simplest case, a vehicle stopped in the middle of freeway would be a good indicator of an abnormal traffic situation.

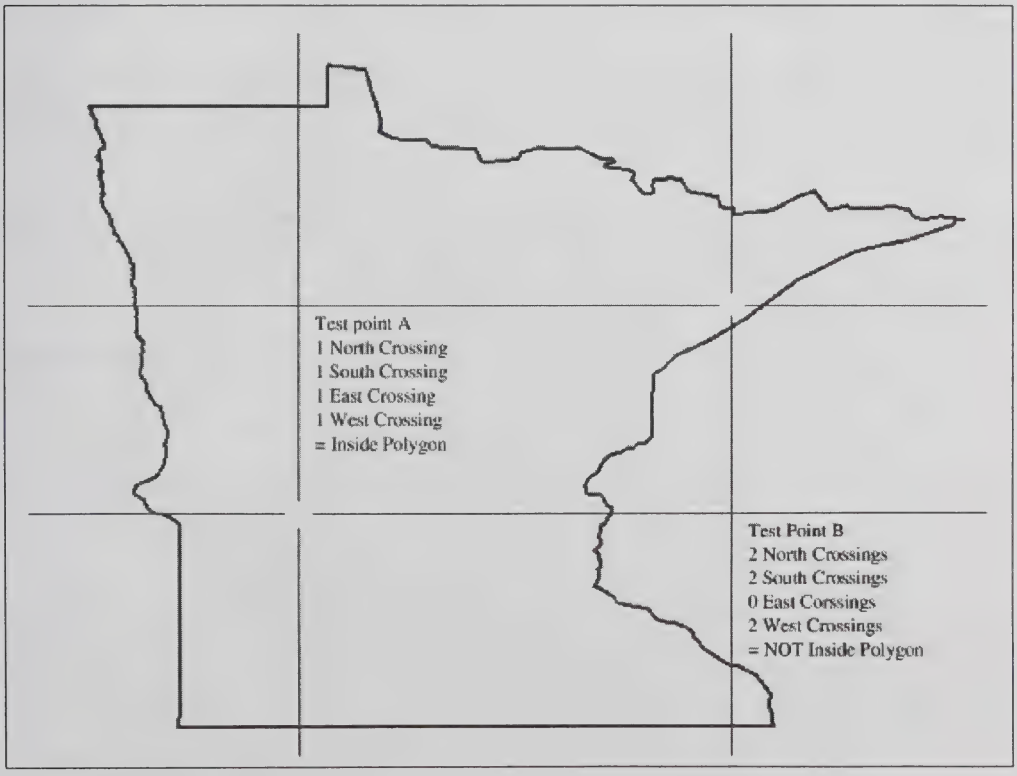
Nearest Road Search



Polygon Searching

I wrote a program called shpdump-poly that can very rapidly determine what polygon a point lies within. This test is performed by drawing an imaginary horizontal and vertical line through the test point. By counting the number of times the line crosses the polygon it is possible to determine if the point falls within or outside the polygon. If, in each of the four cardinal directions, the line crosses the polygon an odd number of times then the point is within the polygon. In order to optimize processing time I load each vertex of the polygon and the associated bounding box into memory. Because bounding box checking is more efficient then point in polygon testing, each test point is first checked to see if it falls within the bounding box. If the point falls within the bounding box it is passed to the slower point in polygon test to determine if it is really in the polygon.

While working on shpdump-poly I hypothesized that performance increases could be accomplished by pre-sorting the polygons so the shapes with the highest likelihood of containing the polygon would be searched first. My hypothesis appeared to be correct because when I implemented pre-sorting based on population I was able to decreased execution time by 7%. Subsequent analysis shows that the correlation between the population of each state in the United States and the number of packets originating from that state is r=0.837243. On a Pentium III class machine, the program can place 500,000 latitude / longitude points into a their corresponding state of the United States in 15 seconds.



Map of Observed Average Speeds for Los Angeles

The map to the left shows the average of speed vehicles traveling in Los Angeles County. Road segments that are colored a shade of red represent sections where data has been collected. The brighter the shade of red the faster the observed speed. This map is based on data I collected from approximately 200 mobile APRS stations over a three month period.

1) My database was populated with three months of data from APRS stations worldwide. Data was collected using the APRS parser program. Three months of data represent approximately 54,000,000 packets stored in the database.

2) Packets from moving APRS stations in the vicinity of Los Angeles County were extracted from the database using this SQL query:

> *SELECT latitude, longitude, packet_id FROM position WHERE speed>=10 AND (latitude >= 32.806 AND latitude <= 34.823) AND (longitude >= -118.945 AND longitude <= -117.649) ORDER BY latitude;*

The WHERE clause of the query limits the data to stations moving more than nine kilometers/hour and that fall within the bounding box around Los Angeles. In this case there were approximately 135,000 data points that met these criteria. Each data point consists of the latitude, longitude, and database packet identifier

3) Because the bounding box around Los Angeles allows some packets from stations outside of Los Angeles, the data is sorted by county using the polygon searching algorithm. Using shpdump-poly the data is sorted to one file per county. This further sorting reduced the data points to approximately 106,000.

4) Once the data is separated, each latitude / longitude point is run through my nearest road searching algorithm, shpdump-arc, to find the closest road. For each data point where shpdump-arc finds a corresponding road, a single line is printed. That line is comprised of the original search latitude, longitude, packet identifier, and a unique identifier for the road segment.

5) The speed at each data point is retrieved from the database based upon the unique packet identifier. This query would retrieve a single speed:

> *SELECT speed, source FROM position WHERE packet_id=35851920;*

Once the speed is determined a SQL INSERT is performed that puts the new record into the database:

> *INSERT INTO speeds (tlid, speed, source, packet_id) VALUES (141792250, 94, "N6NKU", 35851920);*

6) The important database tables are now the speed table containing the observed data points, and the roads table containing the name and unique identifier of every road segment on the map. To find all of the road segments where data has been observed, I use an equi-join between the two tables, based on the road segment. This SQL query finds all of the roads in Los Angeles County and it returns the minimum, maximum, average, and standard deviation of the observed speeds:

See a full-size version of this map at: **http://db.aprsworld.net/mapserver/la.php**

SELECT speeds.tlid, AVG(speeds.speed) AS "speedAvg", MIN(speeds.speed) AS "speedMin", MAX(speeds.speed) AS "speedMax", STD(speeds.speed) AS "speedStddev", COUNT(speeds.speed) AS "num" FROM speeds, roads WHERE roads.fips=06037 AND speeds.tlid = roads.tlid GROUP BY speeds.tlid;

7) For display of the data I use a standard GIS file format called a shapefile. A shapefile consists of three individual files. Two specify the geometry of the shapes and the third file is a database of information associated with each shape. The map of Los Angeles County is a shapefile, however I had to add fields to the shapefile's database to store the speed information for each road segment. I wrote a program, join2dbf, which takes the calculated data from the equi-join above and appends it onto the appropriate record in the map.

After running join2dbf I now have a complete map of Los Angeles that contains the average speed, minimum speed, maximum speed, and standard deviation from each road segment with observed data points.

The map to left will never change, however the programs and techniques that I used to create it could be run continuously to create a dynamic map of speeds. Such a map could be used to identify areas of traffic congestion or used by drivers to avoid slow moving roads.

Results and Conclusions

1) APRS and Automatic Vehicle Location technologies provide data that can be harnessed for many other applications besides simple vehicle tracking. The amount of data generated by these technologies is enormous.

I was able to create tools and techniques for effectively storing and retrieving position information for thousands of vehicles generating millions of pieces of information.

- The database and APRS parser has scaled well and demonstrated its robust nature.
- The datamart web interface provides a convenient means for verifying database integrity and visualizing station locations.
- The polygon searching program is useful for separating data by region or within political boundaries. It can be used as a filter before other data analysis.

2) Automatic detection of traffic congestion has the potential to reduce investments by metropolitan areas in camera systems and public service personnel for monitoring traffic problems.

By observing traffic over a long term it is possible to calculate and map the average speed on roadways. Once such a base map is made, it is possible to detect traffic anomalies, such as traffic jams, by comparing the "live" observed data to previously gathered data.

- The parser that I wrote continuously adds data to the database which has the effect of constantly adjusting the minimum, maximum, and average speeds of the road.

3) Because large fleets of vehicles are being equipped with vehicle locating technologies, there are many new opportunities for automatically collecting near real-time vehicle movement data on a large scale.

Even with a low density of mobile APRS stations, I was able to develop software for observing vehicle movement on many roadways. In the case of Los Angeles, only 200 stations over three months provided

enough data to calculate average road speeds on many highways and some smaller roads. With hundreds of thousands of mobile stations, it would be possible to observe data on many roads with a large number of observations for each road. More observations would also make it possible to compute traffic maps for different time periods during the day, including rush hour versus normal traffic.

Acknowledgements

I would like to thank Dr. Mark Ordal of the Minnesota Academy of Math and Science for his generosity of providing me lab space with a suitable high-speed Internet connection, as well as for the evenings that he stayed late to allow me to keep working. Adam Frisch KB0VYO's Beowulf cluster was responsible for putting the idea in my mind of processing millions of pieces of data. Adam has also spent days working with me to setup computers and keep equipment running. This project would not be possible without the dedicated individuals that have designed and built the APRS networks. I would like to extend a big thank you to the men and women of the open source and free software communities because "If I have seen further it is by standing on ye shoulders of Giants." – Isaac Newton

References

Dimse, Steve. (1997). "APRServe: An Internet Backbone for APRS." Available online: <http://www.aprs.net/aprserve.dcc.html>

Wade, Ian. (2000). " APRS Protocol Specification Version 1.0.1." Available online: <ftp://ftp.tapr.org/aprssig/aprsspec/spec/aprs101/APRS101.pdf>

Created Realities Technology in Amateur Radio

Greg Jones, WD5IVD
Dept. of Technology and Cognition
University of North Texas, Denton, Texas
wd5ivd@tapr.org • http://created-realities.com

Abstract

This paper discusses the possibilities of Created Realities Technology in amateur radio. The paper discusses its use in education/learning, communications, and information presentation. The creation of meaningful 3D spaces for both communications and information presentation can add a new dimension to amateur radios presentation and teaching.

Introduction

The next generation of consumer information presentation will be in three dimensions. A paper by Brock, Kovacs, and Smith entitled "APRS in Hollywood - Integrating Real Time 3D Graphics with Wireless GPS systems" in the 2001 Digital Communications Conference proceedings discussed one aspect of this potential. Their paper discussed the possibilities of integrating real-time APRS systems with real-time 3D computer graphics. The current state-of-the-art in 3D rendering can be provided on currently sold consumer platforms. This approach will become widely available, since each year the number of personal computers equipped with the necessary hardware requirements will continue to expand.

If you have played Doom, Quake, Counter Strike, or numerous console games, then you have interacted with a real-time 3D rendered environment. Over the past ten years, PC gamers who played 3D games spent additional money to enable their computers to support 3D presentations. A typical gamer upgrades their computer almost every year to keep up with the latest gaming. Computer games have traditionally pushed the envelope of real-time computer visual presentation. What has changed in the last few years is that consumer grade computers have finally reached the price-point/feature-set to support real-time 3D rendered graphics. Last year over 70% of windows based personal computers sold contained the necessary hardware and computing power required supporting real-time 3D environments (Jon Peddie Associates, 2001). Older computers (within the last 3-5 years) can be upgraded with a graphics card for less than $100 to allow them to display 3D. Personal computers are not the only platform available. Console games like SonyPlay Station 2 and Microsoft Xbox support 3D graphics and both have support for TCP/IP. This provides over 50 million personal computers and game consoles to examine as possible presentation systems that can communicate using the Internet or radio.

The potential for 3D environments, or what I call 'created realities', is almost endless. After stepping down from TAPR and completing my PhD at the University of Texas Austin, I formed the Created Realities Group in 2001. Our concept has been simple – take current commercial approaches to provide contextually accurate software-derived 3D environments with existing off-the-shelf technology. Combine these created realities with collaborative groupware, unified communications, and other instructional tools to create a single delivery interface running on Windows, Mac, and Linux operating systems for use with education, business, and entertainment.

Distributed Education and Communications

Our current focus has been on creating a distributed education and communication systems. Distributed education can be defined as the acquisition of knowledge and skills through mediated information and instruction. Distributed learning is used in all areas of education including Pre-K through grade 12, higher education, home school education, continuing education, corporate training, military and government training. Research studies have been quite consistent in finding that distance learning classrooms report similar effectiveness results as reported under traditional instruction methods. In addition, research studies often point out that student attitudes about distance learning are generally more positive. Research on distance learning applications for Pre-K through grade 12, as well as in adult learning and training settings, strongly suggests that distance education is an effective means for delivering instruction.

Programming for distance learning with created realities provides the learner with many options both in technical configurations and content design. Educational materials are delivered primarily through live/interactive classes. The intent of our software is to replicate face-to-face instruction by creating a context for instruction using immersive 3D environments. The ability of the teacher and students to see each other may not be a necessary condition for effective distance learning, but audio and contextual reference can be a critical component for interactivity.

Most of today's distributed learning environments are designed to support interaction between one person and a computer with collaboration between multiple users being accomplished using non-immersive tools like electronic mail, shared files, or text-based chat. In non-computerized work settings people interact in a rich environment that includes information from many sources (telephone, whiteboards, computers, physical models, etc) and are able to use these simultaneously and move among them flexibly and quickly (Stanford Computer Graphics Laboratory, 2001). The creation of a useful and integrated virtual classroom for distributed learning has long been an elusive goal. There have been numerous attempts over the years to build user interfaces to deliver realistic environments that create a context for communications, but few have reached wide use and adoption. While the use of virtual on-line communities used for creating collaborative interactions are not new, there has been a scarcity of sophistication in the approaches taken by most educational systems. Ruess, Reed, Gill, and Fusco (2001) stated that reality level suffers for currently deployed environmental based educational systems. Much of the focus for educational delivery systems have been on web based VRML systems, which lack many of the performance qualities that are found in current commercial designs. Created Realities solves these issues and provides a way to more seamlessly provide information flow and increases interactions between participants. Djoudi and Harous (2001) stated in their T.H.E. Journal Online article that:

> Information and communication technologies, on which new training and learning media are based, have improved the transmission and access of data. But they have not facilitated significant interaction between the user and the information. This interaction is very important for someone trying to learn. Until now, new Internet technologies were taught to improve the speed of access and the quantity of information accessed.

How can this be used in amateur radio today ?

Now that we have gotten the basics out of the way, what does this mean for amateur radio. Several things come to mind as things to try first. APRS in 3D seems to be a natural concept to pursue. When the Brock, Kovacs, and Smith (2001) paper was published we had been six months into development of our 3D engine. I read their paper and thought to myself that we could support a real-time interactive system right now. The only problem being that building an APRS display is not on our development road map. If we can find the necessary people to form a working group, then something might be possible. If a programmer wants to do it on their own, they have a variety of pre-existing engines available supporting both OpenGL or DirectX APIs to select from. There are a score of 3D engines that can be licensed today. Engine licenses range from free to very expensive. Since a base-line APRS display system does not need highly advanced graphic flair, then the lower-end engines should be acceptable. Presenting the information in the 3D space is not that complicated. The hard part is creating an efficient means for the server and client to communicate and provide information from the database to all connected users. I would love to talk to anyone interested in developing something like this either using our engine or another approach.

Another concept to pursue is to teach amateur radio classes on-line in real-time. This concept fits into our current strategic roadmap, since we are currently developing for educational distributed learning. A club or individual could hold on-line classes for amateur radio. There are plenty of systems that could support this, but the created realities approach is to provide a single interface that provides all necessary tools for teaching on-line peer-to-peer classes (audio, overheads, e-mail, chat, context, etc). The ability to present amateur radio theory using 3D models could be very interesting. A course could be taught live one time, captured in the database along with all the student interactions, then later delivered to any student that wanted to take the class when it wasn't being offered live. One example of a current project we are developing is to provide on-line contextually accurate language learning (Jones and Squires, 2002). If we can provide language learning on-line, then teaching amateur radio class should be simple.

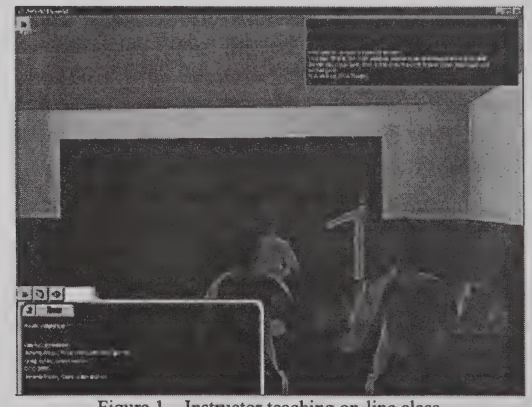


Figure 1 – Instructor teaching on-line class.
(prototype avatars being used)

How about holding virtual amateur radio club meetings. The created realities software would provide an excellent interface for hosting on-line amateur radio meetings. Even the invited guest speaker can do their presentation using audio, overheads, video, etc. We use our software to make presentations and hold work group meetings that include participants from around the world. The bandwidth requirements are low enough to allow participants using 28.8kbps dialup to be involved. One of our goals has always been to deliver low-bandwidth interfaces that can be scaled up to faster connections. Amateur radio interest group could meet on-line to discuss topics or hear presentations. We could even host the ARRL and TAPR Digital Communications Conferences entirely on-line saving the cost of travel :-) Then again meeting people meeting in person and getting away from home has its advantages.

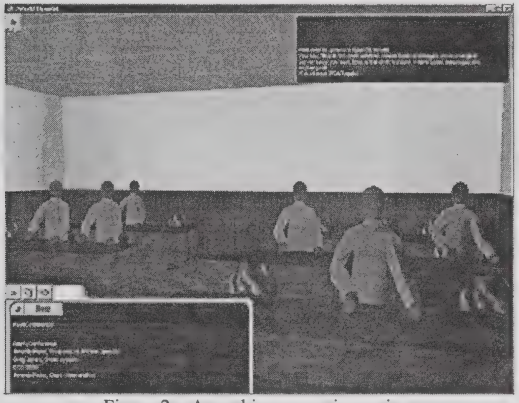


Figure 2 – A working group in session.
(prototype avatars being used)

We have been talking to several museums about creating virtual spaces for museum tours and exhibits (Jones & Christal, 2002). Instead of visiting a museum on-line, how about presenting ham shacks or W1AW ? A ham could have a chat with someone and instead of just hearing their voice come out a speaker, they are actually in the other hams shack – or at least a created version of one. A tour could be given of the W1AW station and the AI (Artificial Intelligence) giving the tour can customize the tour based on feedback received from the participant.

Conclusion

I have touched on just a few of the possibilities of information presentation in 3D space for amateur radio. The use of created realities to present immersive interfaces brings an entirely new level of interaction with it that is beyond 2D web pages or information. The ability to promote active engaged learning and allow a full range of peer-to-peer interactions allows participants to communicate as if they were in face-to-face dialog without the bandwidth requirements of retransmitted systems like video.

References:

Davie, L. E., & Wells, R. (1991). Empowering the learner through computer-mediated communication. *American Journal of Distance Education, 5 (1)*, 15-23.

Djoudi, M, & Harous, S. (2001, November). Simplifying the learning process over the internet. *T.H.E. Journal Online: Technological Horizons in Education.* Retrieved December 20, 2001, from http://www.thejournal.com/magazine/vault/A3738.cfm.

Jon Peddie Associates. (2001). *Software tools and applications series: 3D visualization and simulation market study.* Retrieved August 15, 2001, from http://www.jpa.com/studies/vizsim/index.html.

Jones, J. G. (2002). Enhancing instructor and learner interactions using created realities technologies. Retrieved July 20, 2002, from http://created-realities.com.

Jones, J. G. and Squires, T. (2002). Combining speech recognition and created realities VXInteractive™ system to create authentic spoken language learning. Retrieved July 20, 2002, from http://created-realities.com.

Jones, J. G. and Christal, M. (2002). The future of virtual museums: on-line, immersive, 3D environment. . Retrieved July 20, 2002, from http://created-realities.com.

Communication in Virtual Worlds. Panel session presented at the Vlearn3D 2001, Knowledge Spaces and Information Landscapes virtual conference. Retrieved January 10, 2002, from http://www.vlearn3d.org/conference/panelthree.html.

Stanford Computer Graphics Laboratory. (2001). *Stanford Interactive Workspaces Project.* Retrieved January 5, 2001 from the World Wide Web: http://graphics.stanford.edu/projects/iwork/

Repeater Data Transmission System

Peter Mudie, VK2XZP
pmudie@magna.com.au

Synopsis

The paper describes a specialised data communications system. The data system has the capability of carrying low data rate information over an analogue radio channel at the same time as voice traffic. It is based on the capstone project I completed in order to receive my Bachelor of Engineering Degree[1].

Summary

For many years Radio Amateurs have been developing voice repeaters and their associated links and controllers. Gradually these systems are becoming more and more sophisticated. Unfortunately, much of the modern digital technology is financially beyond the reach of Amateur Radio clubs.

Albeit the technology being developed in this project is not a ground breaking fully digital solution, it provides a bit of the flexibility of digital control over the existing analogue radio links. This is provided at a cost that is affordable and realistic for amateur radio clubs who want to go the step beyond simple linked repeaters.

This system in essence is a FDM (Frequency Division Multiplexed) system, where by the base-band frequency range of 20 Hz to 4.5 kHz is being utilised to carry the voice band modulation. The 5.5-6.5 kHz frequency range is being used to carry 200-300 bit per second data.

This base-band modulation is compatible with many existing narrow band frequency modulated (NBFM) radio links.

[1] The full documentation (with circuit diagrams) is available from www.tapr.org

The purpose of this data link is to allow communication between intelligent microprocessor based repeater controllers at each linked site. This digital communication will carry messages about system mode changes, signal strength information for receiver voting and other telemetry and logging data.

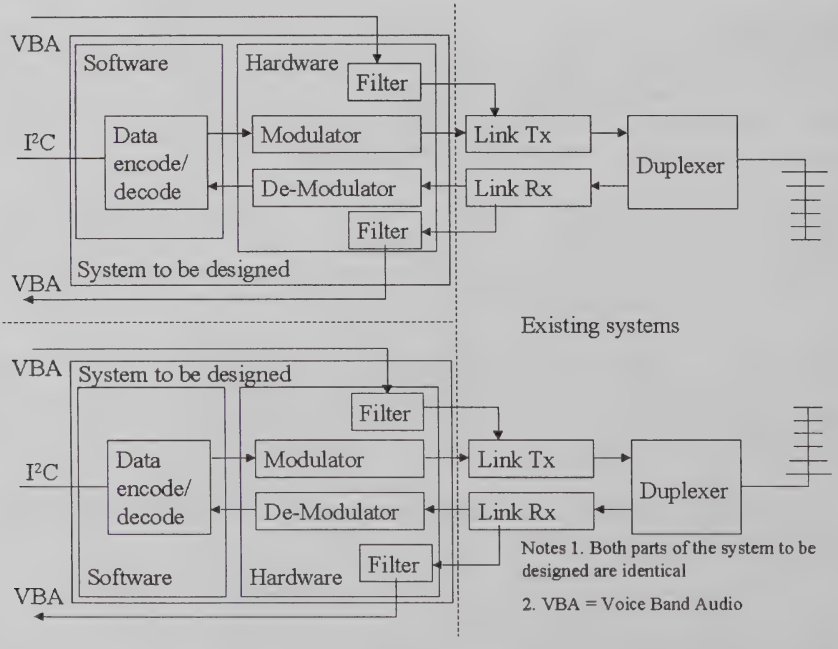
Introduction

This project has a few overall specifications that should be outlined first. The aim is that the outcomes of this project, when appropriately published, will be useable to all Amateur Radio Repeater groups and individuals. With this firmly in mind it must be considered for a moment, the technical ability and purchasing ability of a majority of repeater minded amateurs.

Not many active amateurs are trained specifically in Electrical Engineering by university or TAFE. This immediately suggests that any designs should be sufficiently simple in overall complexity that amateurs in general can build and maintain them.

Most amateurs do not have the range of parts suppliers available to them that a practising Electrical Engineer does. Consequently all parts used where possible should be restricted to those that can be purchased where possible from domestic suppliers, such as Dick Smith Electronics, Jaycar and Altronics.

System Design



The full duplex analogue radio links that the signalling system is conveyed over are in many cases pre-existing infrastructure. Currently these links operate in a simple carrier operated mode. This system will pass data over the links simultaneously with the voice traffic. The system provides an I^2C interface to communicate with the other controllers and an audio interface for connection to the radio communications transmitters and receivers.

The prototypes implement 300 bit per second data rates as this allows testing and debugging of the software using standard serial interfaces on a PC.

Data Types

There are three different types of data to be transmitted over the link. Each type of data requires different handling. The packet types are

1. Time and error sensitive
   These packets will usually be the change in state of a remote receiver's carrier detector. This data needs to be passed quickly (within 100ms) without error, so any packet containing errors will need to be resent.

2. Time sensitive
   These packets will mainly be a value indicating the quality of a remotely received signal that is to be voted. It is essential that these types of packets are passed in a timely manner. They will tend to emanate at about a 20Hz rate when there is voting occurring. If an error occurs the packet in error is to discarded.

3. Error sensitive
   These types of packets are usually mode changes that are user initiated or telemetry data for logging purposes. This type of packet will be divided up into two sub types of packet

   a. The mode changes that need to be transferred within a few seconds (realtime)
   b. Telemetry data for logging that will be transferred only if there is space data capacity or the data channel is idle.

   If errors occur on either of these types of packets, they need to be corrected by a process of retries.

Error control

Error control is required for type 1 and 3 packets. The error detection methods is able to detect at a minimum of one bit error in a packet. This specification has been relaxed for the initial prototype of the software to ensure it is operational in time.

Packet queuing and Retries
The above 3 types of packets need to be queued and transmitted with a priority that is appropriate to their urgency. Packet type 1 has the highest priority, and packet type 3 has the lowest priority.

The software receives commands from the other interfaced microcontrollers and queue them in appropriate order for transmission. Incoming commands are separated into 3 buffers for processing. Retries need to be used if type 1 and 3 packets fail with an error in the data received. The retry is given the same priority as the packet that it is trying to recover. Only type 1 and 3 packets need acknowledgment.

Audio spectrum use

All radios to be used with this system have their audio levels both receive and transmit adjusted such that 0dBm on the baseband inputs and outputs corresponds to peak deviation (in most cases 5kHz deviation).

Spectral content
Voice band audio covers a range of 20Hz to 4.5kHz +0 –3dB Signalling data uses a maximum bandwidth of 1kHz centred on 6kHz. The signalling carrier should not exceed –10dB with respect to the radio systems peak deviation.
Signalling content in voice band audio
All signalling artefacts are greater than 40dB below peak deviation in the demodulated and filtered audio that is presented to the repeater from the link. All voice band audio are greater than 13dB below the signalling carrier level in the signalling bandpass.

Signalling data content in voice band audio post demodulation
Once the Signalling data is recovered from the baseband audio from a link receiver, the signalling data needs to be removed to leave the voice band audio. The maximum level of signalling data to be present in a voice band audio line is at –40dBm.

Hardware

Modulator
Inputs:

- Data stream: 5 V logic

Outputs:

- Signalling data: audio line capable of driving 600 ohm load level to be determined by simulation, suggested range from –30dBm to –10dBm.

Functionality:
The modulator produces an amplitude modulated carrier centred on 6kHz with an

overall bandwidth of no greater than 1kHz. The modulator is to operate at a minimum symbol rate of 200 symbols per second with two amplitude levels.

Demodulator

Input:

- Signalling data: audio line terminating in 600 ohms. Sensitivity of signalling decoder between –30dBm and –10dBm to be determined by simulation. Input signal will contain voice band audio up to 0dBm in level.

Outputs:

- Data stream: 5V logic same sense as input to modulator

Functionality:
The demodulating should be capable of fully recovering the data encoded by the modulator.

Voice band audio filtering and radio interface

The voice band audio filtering is applied at two points in each path. It is applied to the voice band audio path before the signalling data is inserted, and when the signalling data has been recovered from the baseband audio.

This filter needs to pass the audio range 20Hz to 4.5kHz +0 -3dB. There must be greater than 30dB of attenuation across the signalling data band pass of 5.5-6.5kHz. Above 6.5kHz the filter response may rise as high as a loss of 6dB and then it falls off at a minimum of a first order rate.

Protocol

A cost-effective microprocessor from Atmel was chosen for this part of the system. The system will have to communicate with other microprocessors over a bus based communications system. This microprocessor has the task of looking after all the signalling data coding, error correction, and other signalling data house keeping issues.

System Design and Implementation

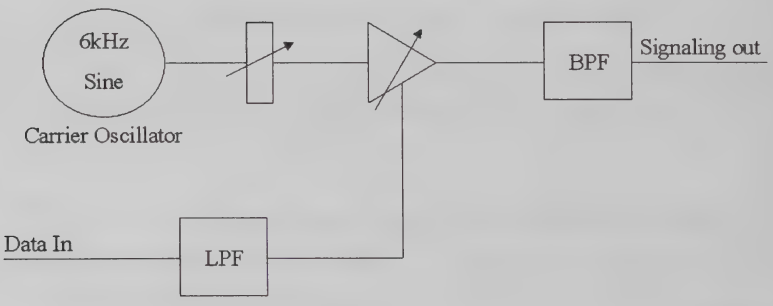
The full duplex analogue radio links that the signalling system will be conveyed over are in many cases pre-existing infrastructure. Currently these links operate in a simple carrier operated mode. This system passes data over the links simultaneously with the voice traffic. The system provides and I^2C interface to communicate with the other controllers and an audio interface for connection to the radio communications transmitters and receivers.

As the specification has stated a data rate of at least 200 bits per second needs to be achieved. The prototypes will implement 300 bit per second data rates as this allows testing and debugging of the software using standard serial interfaces on a PC.

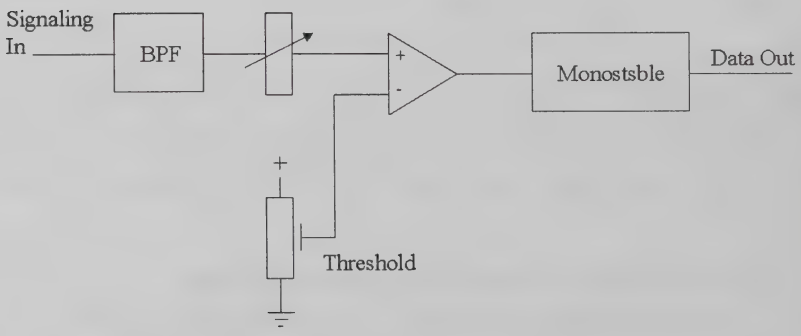
Hardware

The proof of concept prototype provided some confidence in the ability of the radio links to handle the above audible audio signalling. It also provided the basis for the block diagram of the system.

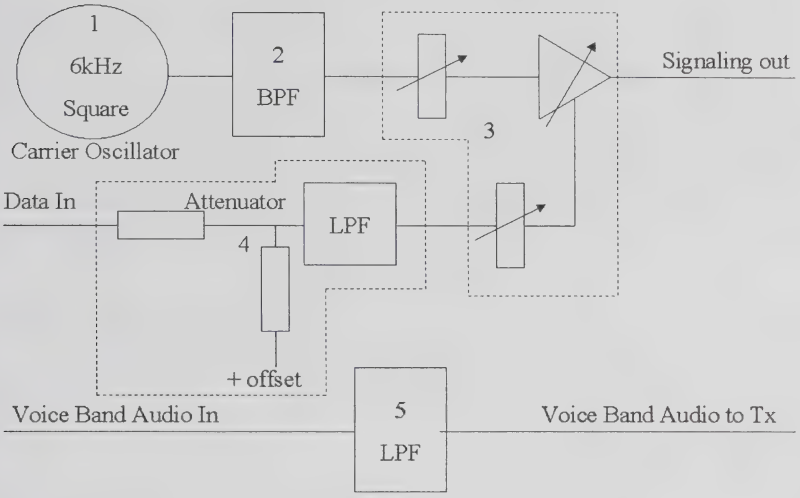
Transmit Modulator - DSB with carrier



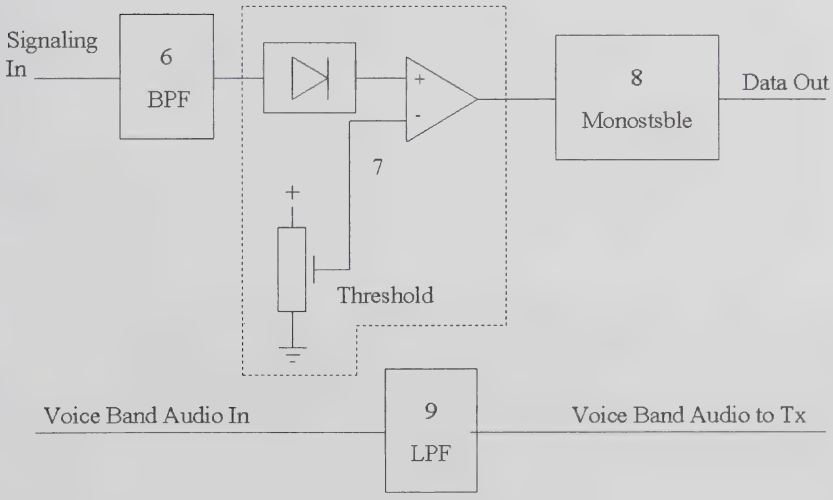
Receive Demodulator - DSB with carrier

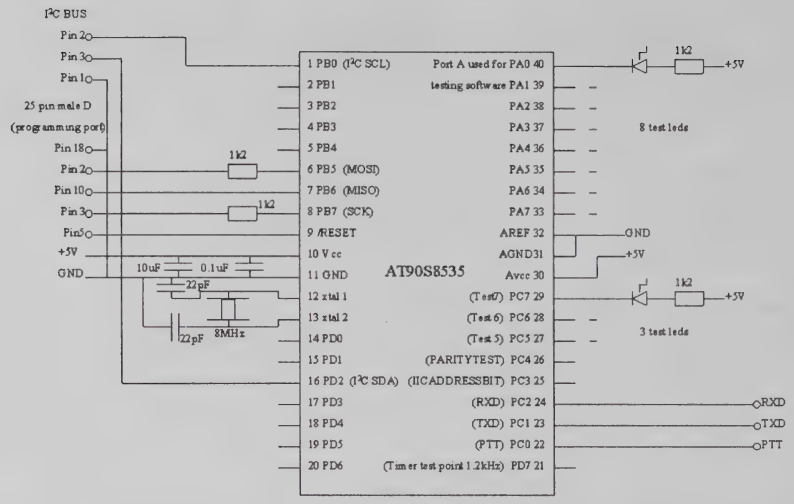


Final Transmit Modulator - DSB with carrier



Final Receive Demodulator - DSB with carrier





Design Realisation

For the implementation phase of the process the actual circuit topologies are determined. As a number of the blocks are audio frequency filters, the topology's being used are pretty well defined and tried and true. The Active filter cookbook was used as a reference on the topic. This provided the normalised value sets to allow for easy scaling of the filters to the desired operating frequency.

The frequency scaling was largely based on fixed capacitor value and obtaining the precision of tuned frequency with the resistor. This was the case as the local hobbyist electronics suppliers have a single value of precision capacitor value available. Dick Smith Electronics stocks a 1nF 1% polystyrene capacitor. Due to a number of the filters being of order 4 or higher, this capacitor was chosen as a necessity to ensure reproducibility. Resistors are easily procurable with a 1% tolerance in the E24 series so these were varied to obtain desired frequency and damping characteristics.

The next step of circuit simulation only applied to the AC based circuits such as the active filters, as the simulator available only support AC analysis. The simulator being used is the ARRL Radio Designer. This simulator, as the name suggests, is aimed at Radio Frequency based circuit analysis, hence the specialisation on AC analysis. This package takes a simple net list entry and allows production of a whole variety of plots. The major plots of interest for this project are the bode-plots and monte-carlo analysis.

The monte-carlo sensitivity analysis reveals whether the filter will meet the specification in a reproducible manner. At this point if non-compliance to the specification is found the design is adjusted and filter simulation repeated

Following the completion of the design stages, the circuit is built to confirm operation. The prototypes were constructed on vero-board. This step reveals if there are any issues not identified in the design and simulation stages.

Result of testing

On the whole the signal processing hardware meets specification. There is one non compliance with specification in the form of the loss at 4.5kHz in the voice band audio filter. This miss of specification was accepted from the design and simulation stage as being an acceptable deviation from the specification. This deviation was due to the additional complexity in meeting specification, compared with one of the projects objectives being to minimise complexity and cost. Despite this the hardware in all sections agreed with the simulated results.

This is an experimental design project, and the final result is a simplified version of the final desired outcome for the longer term. The final outcome involves work levels that exceeded the scope and requirement for this project, hence the specifications were relaxed to allow the initial functional system to be created within the time frame of this project.

Conclusion

This project provided a good mix of work that was hard, challenging and problematic for me as well as some straightforward parts. The hardware components of the project, provided for a well executed and on time section of the project. The hardware area is where I have had most of my experience in the past, so this part of the project came to me pretty easily.

The software sections provided a vast number of problems, and completely destroyed any chance of achieving the desired time frames and time line of this part of the project. I also learnt the most from this section of the project. The number of hidden problems that came to the surface meant that the software work was definitely a very good learning exercise.

As this project was always an experimental design that is not producing a finished product within the scope of the capstone project, there is room for further work beyond the scope of the capstone project. Some of this work includes, further iteration and fine tuning of the voice band audio filter design. There is also room for expanding the behaviour of the software to be a little more intelligent with regard to the traffic being handled and the error correction systems. For example, this project implemented simple single bit parity error checking.

On Amateur Radio Use of IEEE 802.11b Radio Local Area Networks

Paul L. Rinaldo, W4RI
Manager, Technical Relations
American Radio Relay League
w4ri@arrl.org

John J. Champa, K8OCL
Chairman, High Speed Multimedia Working Group
American Radio Relay League
k8ocl@arrl.net

The 2400-2450 MHz band is not only an amateur allocation but is used by other services. Primarily, it's an industrial, scientific and medical (ISM) band with a center frequency of 2450 MHz. Other users of the band must accept any interference from ISM emitters. The amateur service has an allocation in this band that differs somewhat in various countries. However, in the United States, the amateur service has a secondary allocation in the 2400-2402 MHz segment, primary in 2402-2417 MHz and secondary at 2417-2450 MHz. This band (actually a larger band of 2400-2483.5 MHz) is used by a number of unlicensed low-power devices, such as cordless telephones and radio local area networks (RLANs). These include IEEE 802.11b and Bluetooth. Quite often, the trade press mischaracterizes this band as "unlicensed spectrum," which may indicate they're not aware of the amateur primary or secondary allocations. One other thing about amateur allocation status is that, on petition from ARRL, the FCC has issued a Notice of Proposed Rule Making to upgrade the band 2400-2402 MHz to primary status.

This jumble of allocations and uses can be viewed as a glass half empty or half full. One view is that the FCC has loaded this band with so many applications as to make amateur operation very difficult. And, there are growing anecdotal stories that amateur systems, particularly amateur television repeaters, operating in this band are experiencing harmful interference from IEEE 802.11b devices. There is growing use of 802.11b in building and campus RLANs. The general experience is that RLANs inside buildings usually do not radiate much energy outside because of outer wall attenuation. Even windows can attenuate the signal through application of sun shielding film.

The Interference

So, the main interference from 802.11b to amateur systems seems to be the outside RLAN access points (APs). Most operate within the FCC Part 15 Rules and may or may not be a problem to amateur systems depending on proximity, line-of-sight and other factors. Those close by, perhaps with directional antennas bore-sighted toward an amateur station are likely to be a problem. In addition, there are an increasing number of APs operating outside the Rules.

The FCC is aware of some of these high-power APs and is considering enforcement

action. ARRL has a program called Amateur Radio Interference Assessment (ARIA) that is trying to measure the noise level in the 2400-2450 MHz band (and others). However, this is a moving target and the situation could change dramatically in a year.

The Opportunity

802.11b presents the Amateur Radio community with an opportunity to use the inexpensive RLAN cards for high-speed multimedia applications including streaming television. While most prices presently hover around $100, some are available at about half that price. The APs, however, are more expensive by virtue of lower sales volumes but are available for several hundred dollars.

Frequencies

802.11b channels are specified on center frequencies 5 MHz apart:

Channel	Center Freq MHz	Comments
1 2 3 4 5 6	2412 2417 2422 2427 2432 2437	These channels are used in the US and other countries by 802.11b devices. Their emissions fall within the 2400-2450 MHz amateur band.
7 8 9 10 11	2442 2447 2452 2457 2462	These channels are used in the US and other countries by 802.11b devices, but cannot be used in the amateur service.
12 13	2467 2472	Not used by 802.11 in the US.
14	2484	Japan only.

The existence of 14 channels does not mean that all are usable. In fact, the channel bandwidth is 22 MHz, or 11 MHz either side of the center frequency. So channel 1 occupies from 2401 to 2423 MHz. Furthermore, the receivers are such that there should be 25 MHz separation between channel centers used in the same location. In practice, channels 1, 6 and 11 are the popular ones.

Channels 1 and 6 fall completely within the amateur band 2400-2450 MHz, and could be used by amateurs under Part 97 of the FCC Rules permitting spread spectrum (SS) operation. In fact, amateurs are permitted up to 1 watt of transmitter power output without automatic power control (APC) and up to 100 watts if APC is used.

Band Plan

The existing (1991) ARRL band plan for the 13-cm band was not written with 802.11b in mind. Here are the existing band plan segments:

2400-2403	Satellite
2403-2408	Satellite high-rate data
2408-2410	Satellite
2410-2413	FM repeaters (25 kHz spacing) output
2413-2418	High-rate data
2418-2430	Fast-scan TV
2430-2438	Satellite
2433-2438	Satellite high-rate data
2438-2450	Wideband FM, FSTV, FMTV, SS, experimental

It is not possible to pick an 802.11b channel within the 1 to 6 range without bumping into another use specified in the band plan. Bear in mind, however, that there may be some local variations. While it is good practice to operate within the applicable band plan, some flexibility exists. Generally, an amateur station operating in accordance with a band plan has some precedence over a station not operating according to the band plan. The main thing is to avoid harmful interference to users operating in accordance with the

band plan. The ARRL *Repeater Directory* lists some, but not all, uses of the band. The local repeater coordinator should have additional information of who is doing what in order to avoid interference to existing users.

.

Amateurs in Livingston County (MI) are in the process of planning what might be the first amateur 802.11b network. They are coordinating their experiments with the ARRL High Speed Multimedia Working Group (HSMM) and the Michigan Area Repeater Council (MARC).

Current plans call for using 802.11b Channel 6 with a center frequency of 2437 MHz. This approach will place the 22 MHz spread spectrum signal in what appears to be the most logical frequency for such testing. Approximately half of the signal is in the experimental portion of the band (2438–2450 MHz) already designated for spread spectrum use. The other half of the signal is in the currently un-used satellite sub-band (AMSAT-OSCAR 40 downlinks around 2401 MHz) and the 2.4 GHz fast-scan ATV sub-band.

If effective APC techniques can be developed, the experimenters plan to use RF output power in the range of 2-4 watts. With small dish antennas and helical beams, the experimenters hope to achieve throughputs in the range of 1-3 Mbit/s over a range of 10 miles or more.

Identification

An amateur station using 802.11b must identify periodically according to Part 97. Some have considered modification of the 802.11b protocol to map station call signs into the frames in a manner similar to that used in AX.25.

The simplest way, at least for now, is to identify in the text of the message, so that anyone with a normal 802.11b card can read the identities of the transmitting stations.

In the Livingston County amateur experimental high-speed network mentioned previously, identification will be call signs typed in normal 802.11b text. Normal voice identification will use streaming audio. Normal ATV identification will be used for streaming video. If you would like more details on this experimental amateur high-speed multimedia network, please contact one of the authors.

Interoperation with Part 15 RLANs

This is very sticky. Technically, an amateur station using 802.11b could interoperate with an RLAN operating under Part 15 rules. However, communication between FCC Parts is considered a "no-no." Nevertheless, it's possible for an amateur using the same 802.11b card to communicate with an RLAN under Part 15 of the Rules.

The problem is that a message received over a Part 15 link must be screened for permissible content before it can be introduced into a Part 97 link. However, a proper Part 97 message could be sent on a Part 15 link.

Confusing? The Rules were not written with any of this in mind.

Interference Issues

While the amateur services have primary and secondary allocations in the 2400-2450 MHz band, and the Part 15 devices operate there without license and must not interfere with or claim protection from licensed services, there is a problem. The amateur licensee looks at the regulatory status and sees there is no need to take Part 15 devices into account before transmitting. If interference from Part 15

devices is harmful, the amateur can report this to the FCC for enforcement action.

On the other hand, the Part 15 user just laid out $100 or so for the RLAN card and expects it to operate. He or she may not be happy that it doesn't work and may become convinced that the cause is interference from an amateur station.

Unfortunately, there are more of them than there are of us. A large number of complaints coming to Washington could be a problem and the outcome might not be what we want. Therefore, it pays to take this seriously and consider mitigation of interference to/from RLANs in any amateur network.

Best Practices

There is a need to develop a set of best practices in use of 802.11b in the amateur service. This paper touches on a number of considerations.

Recommended Additional Reading

Wireless LANs End to End, Bruce III, Walter R., and Gilster, Ron (Series Editor), Hungry Minds, Inc., 2002.
http://www.hungryminds.com/
ISBN: 0-7645-4888-3

Microwave Handbook, Volume 1, Components and Operating Techniques, Dixon, M.W.-G3PFR (Editor), Radio Society of Great Britain, 1989. ISBN 0-900612-89-4
http://www.arrl.org/

Also see:
http://www.qsl.net/kb9mwr/projects/wireless/plan.html.

Current FCC Regulations

FCC Part 97.311 Spread Spectrum Rules apply to 802.11b

97.311 SS emission types

(a) SS emission transmissions by an amateur station are authorized only for communications between points within areas where the amateur service is regulated by the FCC and between an area where the amateur service is regulated by the FCC and an amateur station in another country that permits such communications. SS emission transmissions must not be used for the purpose of obscuring the meaning of any communication.

(b) A station transmitting SS emissions must not cause harmful interference to stations employing other authorized emissions, and must accept all interference caused by stations employing other authorized modes.

(c) When deemed necessary by a District Director to assure compliance with this Part, a station licensee must:

(1) Cease SS emission transmissions;
(2) Restrict SS emission transmissions to the extent instructed; and
(3) Maintain a record, convertible to the original information (voice, test, image, etc.) of all spread spectrum communications transmitted.

(d) The transmitter power must not exceed 100 W under any circumstances.
If more than 1 W is used, automatic transmitter control shall limit output power to that which is required for the communication. This shall be determined by the use of the ratio, measured at the receiver, of the received energy per user data bit (Eb) to the sum of the received power spectral densities of noise (N0) and co-channel interference (I0). Average transmitter power over 1 W shall be automatically adjusted to maintain an Eb/(N0+I0) ratio of no more than 23 dB at the intended receiver.

Emergency Radio Email (ER-Email)

Paul Schreiber, W2UH

This description of ER-Email is submitted to the Digital Communications Conference to solicit from this pool of experts (1) a realistic assessment its potential as an important emergency communications mode, (2) the effort required to develop it, and (3) what group or individuals are willing to develop it.

I feel very strongly that Emergency Radio-Email has the potential of becoming an important ARES/RACES emergency communications mode with a very high degree of acceptance among served agencies. The name was chosen to quickly convey to served agencies what ER-Email will do for them.

Acceptance

Widespread use of Internet messaging plus a review of the acceptance of packet by our served agencies in drills since 1988 convinces me that there will be less "keyboard fright' than mike fright among the personal we serve. They are no longer surprised by errorless text messages, *they will expect them*. Why else does AOL feature instant messages in its TV ads?

Advantages

Served agencies will highly value these features of ER-Email:

- A high degree of security, especially Red Cross Chapters.
- Errorless text messages.
- Hard copies of these messages.
- Date and time stamps, especially Offices of Emergency Management.
- All messages saved to disk

ARES/RACES emergency communicators will benefit from these features of ER-Email

- Elimination of constant monitoring as with voice nets.
- Elimination of a net control operator.
- Much less on-the-air time for each message.
- Immunity to local audible noises.

Basic outline of ER-Email

To minimize operator training and increase the comfort of personnel at served agencies the most important element of ER-Email is a PC window very closely mimicking popular Internet browsers. The software includes a mailbox for incoming messages. For universal compatibility AX-25 packet output comes from either an internal sound card or a small external TNC.

ER-Email window

In overall appearance the window must look and feel like email windows of popular browsers. Email buttons and dropdown menus include **You have mail**, **Create mail, Send mail, Read mail,** and **Print mail**. Dropdown menu for **Offline mail** includes **Mail you've read, Mail you've sent,** and **Mail waiting to send**. Buttons specific to packet are **My call, My alias**, and **Settings**. **Settings** dropdown menu has all the more exotic packet parameters for experienced packeteers to worry about, and keep hidden for non-packeteer operators.

An **Address Book** lists other packet stations in the emergency by a full tactical description plus the shorter tactical alias used in the ER-Email address field. This allows changes in My Call with new operators. For instance, the Middle School Shelter listing has an alias MIDSCH.

After clicking on **Send email** popup messages indicate; **Connecting to [*], Connected to [*], Sending email to [*], Email [subject] received by [*],** or **Connection lost .** A **[*]** is full name of addressee, for instance, Middle School Shelter.

All received and sent mail is date and time stamped, and automatically saved and retrievable using the buttons mentioned above.

Mailbox

ER-Email packets typically go to target station's mailbox, and **You have mail** button brightens. This operational mode more closely mimics Internet email and eliminates need for constant operator monitoring. A **Chat** function allows keyboard-to-keyboard conversations if desired.

AX-25 packets

For minimum extra hardware, AX-25 packets come from an internal sound card or small external, battery powered TNC. One of my ARES/RACES operators, Andy Stillinger, WA2DKJ, once crammed a TNC into a connector shell!

Digipeaters, not PBBSs or networks

To reduce the need for additional packet infrastructure, digipeaters connect out-of-range stations. Checking a PBBS for messages is eliminated. Some incidents may need a dedicated digipeater at a high location. Otherwise stations in the net also digipeat. The Address Book includes the suitable digipeater for each target station.

Date Rate

Data rates of 1200 Baud are completely acceptable. Only text messages of a less than a page are expected. One-page official damage assessment reports have been transmitted by packet at speed comparable to faxes.

Voice link

As envisioned, each packet station also has a voice link for quick tactical exchanges. The voice operator is also the control operator for the packet station. If desired, this allows an unlicensed person, preferably personnel from the served agency, to send and receive ER email.

Printers

Printers, now available for $100 or less, are important for hard copies, especially at control sites such as EOCs, Red Cross Chapters, and shelters. This may be a problem when power is lost.

My credentials

I've served as the ARES/RACES director of Chatham Borough and Chatham Township (New Jersey) for over 20 years. Packet was installed at both Emergency Operating Centers after a packet demonstration to officials of both communities in 1988. Since then we used packet in at least 10 drills. Acceptance is especially high at the Southeast Morris Chapter of the Red Cross, which serves the Chathams.

ER-Email was conceived after a countywide packet drill on November 5, 2001. The post-drill critique revealed the need for a much more user-friendly packet program. Even our formerly active packeteers had become rusty using their favorite programs (we used five different programs) because we all now use the Internet. What would be easier, for both hams and our served agencies, than a packet program that mimics Internet email?

O. Paul Schreiber, W2UH
33 Ellers Drive
Chatham, NJ 07928-2218
973-635-1290
otmarpaul@cs.com or w2uh@arrl.net

802.11 and Ham Radio

Darryl Smith, VK2TDS
POBox 169
Ingleburn NSW 2565
Australia
Ph: +61 412 929 634
Darryl@Radio-Active.Net.Au

Introduction

The price of 802.11b equipment is continuing to fall. Once retailing for hundreds of dollars, PCMCIA 802.11b cards are now retailing for under $50. These cards contain sophisticated hardware and software that rivals most of the digital technology used in the Ham Radio world today.

A hobby has grown out of this technology in computing circles – building antennas and whole networks, operating at high speeds, with little knowledge of sound engineering principles. Antennas that do not resonate and N Connectors crimped with a pair of pliers are just a couple of the examples of the state of the hobby outside the ham radio world.

If Ham Radio is to survive in the coming years, it is areas such as 802.11b that can be used not only to attract those interested in digital communications, but also as a building block for our own networks.

In this paper I will describe some of the technology, and some of the areas that Ham Radio operators can investigate in order to extend the state of the art.

Frequency of Operation

802.11b is based on Direct Sequence Spread Spectrum utilizing 22 MHz channels centered from 2.412 GHz to 2.462 GHz. There are only three channels orthogonal channels available. Another two channels can be used, subject to acceptance of some interference . These devices are licensed under FCC Part 15.

Contrast this to the Ham Radio allocation in the USA contains an allocation from 2.390 to 2.450 MHz, allowing channels centered on 2.412 to 2.437 (Channels 1-6) to be used under FCC Part 97 (Ham Radio) rules.

Whilst Part 15 allows commercial use, it does this with limitations on the effective transmitted power. The power limit is increased under Part 97, but with the requirement for power control under certain circumstances, and for non-commercial use.

In essence, existing 802.11b hardware can be used either under the Part 15 rules, or under Part 97 rules, allowing

this technology to be used as a building block for commercial and non-commercial applications.

Experimentation

802.11b equipment lends itself to experimentation. Several avenues for experimentation exist

- Antennas
- Protocols/Routing
- Hardware Modifications (Frequency/Power)
- Ethernet/USB up the antenna
- Applications (Digital Voice)

Antennas

One of the great areas of experimentation with 802.11 technology is the design and manufacture of antennas to increase the range of the units whilst still operating within the license conditions. Three types of antennas are popular in the 802.11 experimenters world.

- Pringles Can
- MDS Antennas
- Vertical Antennas

Some of the antennas being used have distinct problems, such as not being able to resonate efficiently. When combined with the use of 30 feet of RG-213 in excess of what is needed, poor results are experienced. However due to the error controls and link margins in the system, the link might actually work.

Pringles Cans

Probably the best example of experimentation with 802.11b is the Pringles can antenna, where a Pringles can is fed with a short stub, and a number of washers are placed on a piece of steel inside the can. Simple analysis shows that this antenna will not resonate correctly at the low end of the band. This is partially offset by the use of washers inside the can creating an antenna that could be best described as an inverse cavity antenna.

MDS Antennas

MDS, or Microwave Distribution System antennas are popular in areas where the cable for Cable TV is not present. It reduces the economic investment for an operator wanting to get into Cable TV significantly.

In Australia the major MDS company went bankrupt a few years ago, leading to a large number of the Conifer antennas turning up on the 2nd hand market.

Operating at about 18 dBi, these antennas are actually quite effective, and there is little that can be done to cause these antennas not to cause them not to work.

Vertical Antennas

Several collinear antenna designs are available on the internet, although many of them have problems which lead to less than desirable results. With some effort these designs could be optimized to allow construction to broader tolerances.

Protocols and Routing

One area that hams should be able to work well on is designing intelligent protocols for 802.11 networks and systems. Routing protocols exist for fully or mostly wireless 802.11 networks, but none have a large following in the field.

Combining some of the ideas contained in the Radio Shortest Path First protocol with traditional wired protocols could yield some promising results. The dynamics of mesh networks do not tend to be as well understood as wired networks.

Hardware Modifications

One of the problems when attempting to build high speed radio-communications equipment is the RF side. Using 802.11b units as a building block simplifies building equipment.

Several options exist for modifying 802.11b units, provided that they are to be used under Part 97. As discussed earlier Part 97 allows higher power outputs, alternate frequencies and higher antenna gains.

Increased Power Output

When operating under Part 97, 802.11b equipment can operate at higher powers. Amplifying the signal is a challenge, since half duplex communications are used on a single frequency. Due to the symmetrical nature of the system it is also no use just amplifying one end of a link. Both ends will need to be amplified.

The question is how do we increase power? There are a few options which I will now discuss.

The first is to find a unit that can be programmed in software to use a higher power, such as the LinkSys WAP-11, which can transmit up to 100 mWatts.

Another option is to place an amplifier external to the 802.11b unit. In order for the amplifier to work, it needs to sense transmit power on the input to the amplifier, and only amplify the signal when power is present, and bypassing the amplifier in receive mode.

Whilst this sounds easy, the amplifier must have VERY fast switching times, which may be difficult to realize in practice. One group has reportedly produced a design for this.

One more option involves more research. Many 802.11b units have circuit diagrams available on the FCC web site. Examining these circuit diagrams will show where the power amplifier is inside the unit, allowing a larger device to be installed.

Alternately the transmit/receive switch line could be identified from the circuit diagram, and use to drive an external amplifier.

Frequency Change

Many 802.11b devices use chipsets that use a couple of frequencies internally. The chipset used on one device from D-LINK uses a reference oscillator, and a separate local oscillator. Changing the

frequency of operation is almost as simple as changing the frequency of the local oscillator.

Of course that will only move the frequency within a relatively small range. In order to change the frequency more than that, more drastic changes are needed. The same unit from D-Link has a separate mixer device – combined for transmit and receive.

With some effort the mixer can be changed for an external unit operating at almost any frequency.

Another option is to use a transverter, operating in a similar manner to the carrier sense amplifier mentioned about. In this case 2.4 GHz is used as an intermediate frequency. Since a transverter contains a power amplifier, problems inherent in power amplifiers added to 802.11 also exist in transverters.

Ethernet up the Antenna

Ethernet up the Antenna is the Holy Grail of almost every computer literate ham. Cat-5 Ethernet cable is much cheaper than Belden 9913, with significantly less loss.

Putting active devices up the antenna allow the distance between the 802.11b device and the antenna to be so small that even RG-58 could be used without serious losses.

The main point to watch is surge protectors for lightning on the incoming Ethernet cable. To a certain extent 802.11b equipment is disposable, but that does not apply to computer systems.

Unfortunately the cheap devices tend to be not Ethernet, but USB. This is not a problem, since USB cables can be connected up to 25m from the computer. In order to get this far away, Hubs or extension cables are needed.

Applications

We have now seen how 802.11b can used, or modified for use by hams to give us bandwidth. The question then becomes 'How can we use this bandwidth?'.

Some answers to this could be

- Digital Video (ATV)
- Digital Audio for repeater linking
- Digital Audio

Digital ATV

Most parts of the world are using high definition signals with complex modulation schemes for Digital TV. These are far too expensive at the moment except for those working in the television industry to experiment with most receivers being quite rare.

This does not lend itself to experimentation. What does lend itself to experimentation is an MPEG encoded video stream transmitted on an 802.11 transmitter. The cost of equipment is small, particularly compared to the average ATV setup.

All that would be required for this type of setup is a cheap WebCam, computer and 802.11 unit with a good antenna. The 802.11 unit could be removed from

the equation if the repeater site has a high speed data connection.

Some work would be required to implement this since multi-cast protocols would need to be used, but this is an area that could see some experimentation.

Digital Voice

802.11 is appearing in consumer handheld equipment such as Palm Pilots. A cute application would be to turn one of these units into a HT. Voice signals connect to the local access point, and get forwarded to an IRLP repeater.

Proximity APRS

In association with Digital Voice is another mode, which I am calling Proximity APRS. This is almost identical to normal APRS, but it based on the access point that is being used by the equipment, rather than GPS position. As a person moves, so does the access point being used, allowing interesting applications. Combining this data with the APRS data stream would not be too difficult.

Digital Audio Repeater Linking

Many parts of the world have complex repeater linking systems. New Zealand has a repeater system that spans the whole country. With IRLP, a world wide repeater system is becoming a possibility.

Many repeater sites do not have Internet access, or the owners have decided not to join IRLP, but want to connect their repeaters together anyway.

802.11 provides a possible solution. The bandwidth available makes it possible for many channels of high quality audio and signaling information to be transmitted on the same frequency.

Imagine a repeater voting system that contains a multitude of receivers and transmitters along a highway. A system could be designed where the received signals from all the sites are combined in a DSP chip to obtain the best signal, regardless of fading. The DSP would have access to all the audio signals so could cut and paste' at will.

Conclusion

What I have shown in this paper is that there are non-traditional sources for equipment that the modern Ham Radio operator can use as part of their hobby. I have outlined some areas for experimentation, and some of the applications that used with the technology.

A Software-Defined Radio for the Masses, Part 1

This series describes a complete PC-based, software-defined radio that uses a sound card and an innovative detector circuit. Mathematics is minimized in the explanation. Come see how it's done.

By Gerald Youngblood, AC5OG

A certain convergence occurs when multiple technologies align in time to make possible those things that once were only dreamed. The explosive growth of the Internet starting in 1994 was one of those events. While the Internet had existed for many years in government and education prior to that, its popularity had never crossed over into the general populace because of its slow speed and arcane interface. The development of the Web browser, the rapidly accelerating power and availability of the PC, and the availability of inexpensive and increasingly speedy modems brought about the Internet convergence. Suddenly, it all came together so that the Internet and the worldwide Web joined the everyday lexicon of our society.

A similar convergence is occurring in radio communications through digital signal processing (DSP) software to perform most radio functions at performance levels previously considered unattainable. DSP has now been incorporated into much of the amateur radio gear on the market to deliver improved noise-reduction and digital-filtering performance. More recently, there has been a lot of discussion about the emergence of so-called software-defined radios (SDRs).

A software-defined radio is characterized by its flexibility: Simply modifying or replacing software programs can completely change its functionality. This allows easy upgrade to new modes and improved performance without the need to replace hardware. SDRs can also be easily modified to accommodate the operating needs of individual applications. There is a distinct difference between a radio that internally uses software for some of its functions and a radio that can be completely redefined in the field through modification of software. The latter is a software-defined radio.

This SDR convergence is occurring because of advances in software and silicon that allow digital processing of radio-frequency signals. Many of these designs incorporate mathematical functions into hardware to perform all of the digitization, frequency selection, and down-conversion to base-

8900 Marybank Dr
Austin, TX 78750
gerald@sixthmarket.com

band. Such systems can be quite complex and somewhat out of reach to most amateurs.

One problem has been that unless you are a math wizard and proficient in programming *C++* or assembly language, you are out of luck. Each can be somewhat daunting to the amateur as well as to many professionals. Two years ago, I set out to attack this challenge armed with a fascination for technology and a 25-year-old, virtually unused electrical engineering degree. I had studied most of the math in college and even some of the signal processing theory, but 25 years is a long time. I found that it really was a challenge to learn many of the disciplines required because much of the literature was written from a mathematician's perspective.

Now that I am beginning to grasp many of the concepts involved in software radios, I want to share with the Amateur Radio community what I have learned without using much more than simple mathematical concepts. Further, a software radio should have as little hardware as possible. If you have a PC with a sound card, you already have most of the required hardware. With as few as three integrated circuits you can be up and running with a Tayloe detector—an innovative, yet simple, direct-conversion receiver. With less than a dozen chips, you can build a transceiver that will outperform much of the commercial gear on the market.

Approach the Theory

In this article series, I have chosen to focus on practical implementation rather than on detailed theory. There are basic facts that must be understood to build a software radio. However, much like working with integrated circuits, you don't have to know how to create the IC in order to use it in a design. The convention I have chosen is to describe practical applications followed by references where appropriate for more detailed study. One of the easier to comprehend references I have found is *The Scientist and Engineer's Guide to Digital Signal Processing* by Steven W. Smith. It is free for download over the Internet at **www.DSPGuide.com**. I consider it required reading for those who want to dig deeper into implementation as well as theory. I will refer to it as the "DSP Guide" many times in this article series for further study.

So get out your four-function calculator (okay, maybe you need six or seven functions) and let's get started. But first, let's set forth the objectives of the complete SDR design:

- Keep the math simple
- Use a sound-card equipped PC to provide all signal-processing functions
- Program the user interface and all signal-processing algorithms in *Visual Basic* for easy development and maintenance
- Utilize the Intel Signal Processing Library for core DSP routines to minimize the technical knowledge requirement and development time, and to maximize performance
- Integrate a direct conversion (D-C) receiver for hardware design simplicity and wide dynamic range
- Incorporate direct digital synthesis (DDS) to allow flexible frequency control
- Include transmit capabilities using similar techniques as those used in the D-C receiver.

Analog and Digital Signals in the Time Domain

To understand DSP we first need to understand the relationship between digital signals and their analog counterparts. If we look at a 1-V (pk) sine wave on an analog oscilloscope, we see that the signal makes a perfectly smooth curve on the scope, no matter how fast the sweep frequency. In fact, if it were possible to build a scope with an infinitely fast horizontal sweep, it would still display a perfectly smooth curve (really a straight line at that point). As such, it is often called a *continuous-time signal* since it is continuous in time. In other words, there are an infinite number of different voltages along the curve, as can be seen on the analog oscilloscope trace.

On the other hand, if we were to measure the same sine wave with a digital voltmeter at a sampling rate of four times the frequency of the sine wave, starting at time equals zero, we would read: 0 V at 0°, 1 V at 90°, 0 V at 180° and –1 V at 270° over one complete cycle. The signal could continue perpetually, and we would still read those same four voltages over and again, forever. We have measured the voltage of the signal at discrete moments in time. The resulting voltage-measurement sequence is therefore called a *discrete-time signal*.

If we save each discrete-time signal voltage in a computer memory and we know the frequency at which we sampled the signal, we have a *discrete-time sampled signal*. This is what an analog-to-digital converter (ADC) does. It uses a sampling clock to measure discrete samples of an incoming analog signal at precise times, and it produces a digital representation of the input sample voltage.

In 1933, Harry Nyquist discovered that to accurately recover all the components of a periodic waveform, it is necessary to use a sampling frequency of at least twice the bandwidth of the signal being measured. That minimum sampling frequency is called the *Nyquist criterion*. This may be expressed as:

$$f_s \geq 2f_{bw} \qquad \text{(Eq 1)}$$

where f_s is the sampling rate and f_{bw} is the bandwidth. See? The math isn't so bad, is it?

Now as an example of the Nyquist criterion, let's consider human hearing, which typically ranges from 20 Hz to 20 kHz. To recreate this frequency response, a CD player must sample at a frequency of at least 40 kHz. As we will soon learn, the maximum frequency component must be limited to 20 kHz through low-pass filtering to prevent distortion caused by false images of the signal. To ease filter requirements, therefore, CD players use a standard sampling rate of 44,100 Hz. All modern PC sound cards support that sampling rate.

What happens if the sampled bandwidth is greater than half the sampling rate and is not limited by a low-pass filter? An *alias* of the signal is produced that appears in the output along with the original signal. Aliases can cause distortion, beat notes and unwanted spurious images. Fortunately, alias frequencies can be precisely predicted and prevented with proper low-pass or band-pass filters, which are often referred to as *anti-aliasing* filters, as shown in Fig 1. There are even cases where the alias frequency can be used to advantage; that will be discussed later in the article.

This is the point where most texts on DSP go into great detail about what sampled signals look like above the Nyquist frequency. Since the goal of this article is practical implementation, I refer you to Chapter 3 of the DSP Guide for a more in-depth discussion of sampling, aliases, A-to-D and

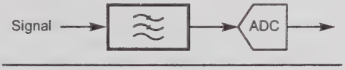


Fig 1—A/D conversion with antialiasing low-pass filter.

D-to-A conversion. Also refer to Doug Smith's article, "Signals, Samples, and Stuff: A DSP Tutorial."[1]

What you need to know for now is that if we adhere to the Nyquist criterion in Eq 1, we can accurately sample, process and recreate virtually any desired waveform. The sampled signal will consist of a series of numbers in computer memory measured at time intervals equal to the sampling rate. Since we now know the amplitude of the signal at discrete time intervals, we can process the digitized signal in software with a precision and flexibility not possible with analog circuits.

From RF to a PC's Sound Card

Our objective is to convert a modulated radio-frequency signal from the frequency domain to the time domain for software processing. In the frequency domain, we measure amplitude versus frequency (as with a spectrum analyzer); in the time domain, we measure amplitude versus time (as with an oscilloscope).

In this application, we choose to use a standard 16-bit PC sound card that has a maximum sampling rate of 44,100 Hz. According to Eq 1, this means that the maximum-bandwidth signal we can accommodate is 22,050 Hz. With quadrature sampling, discussed later, this can actually be extended to 44 kHz. Most sound cards have built-in antialiasing filters that cut off sharply at around 20 kHz. (For a couple hundred dollars more, PC sound cards are now available that support 24 bits at a 96-kHz sampling rate with up to 105 dB of dynamic range.)

Most commercial and amateur DSP designs use dedicated DSPs that sample intermediate frequencies (IFs) of 40 kHz or above. They use traditional analog superheterodyne techniques for down-conversion and filtering. With the advent of very-high-speed and wide-bandwidth ADCs, it is now possible to directly sample signals up through the entire HF range and even into the low VHF range. For example, the Analog Devices AD9430 A/D converter is specified with sample rates up to 210 Msps at 12 bits of resolution and a 700-MHz bandwidth. That 700-MHz bandwidth can be used in under-sampling applications, a topic that is beyond the scope of this article series.

The goal of my project is to build a PC-based software-defined radio that uses as little external hardware as possible while maximizing dynamic range and flexibility. To do so, we will need to convert the RF signal to audio frequencies in a way that allows removal of the unwanted mixing products or images caused by the down-conversion process. The simplest way to accomplish this while maintaining wide dynamic range is to use D-C techniques to translate the modulated RF signal directly to baseband.

[1]Notes appear on page 21.

We can mix the signal with an oscillator tuned to the RF carrier frequency to translate the bandwidth-limited signal to a 0-Hz IF as shown in Fig 2.

The example in the figure shows a 14.001-MHz carrier signal mixed with a 14.000-MHz local oscillator to translate the carrier to 1 kHz. If the low-pass filter had a cutoff of 1.5 kHz, any signal between 14.000 MHz and 14.0015 MHz would be within the passband of the direct-conversion receiver. The problem with this simple approach is that we would also simultaneously receive all signals between 13.9985 MHz and 14.000 MHz as unwanted images within the passband, as illustrated in Fig 3. Why is that?

Most amateurs are familiar with the concept of sum and difference frequencies that result from mixing two signals. When a carrier frequency, f_c, is mixed with a local oscillator, f_{lo}, they combine in the general form:

$$f_c f_{lo} = \frac{1}{2}[(f_c + f_{lo}) + (f_c - f_{lo})] \qquad \text{(Eq 2)}$$

When we use the direct-conversion mixer shown in Fig 2, we will receive these primary output signals:

$$f_c + f_{lo} = 14.001\ \text{MHz} + 14.000\ \text{MHz} = 28.001\ \text{MHz}$$
$$f_c - f_{lo} = 14.001\ \text{MHz} - 14.000\ \text{MHz} = 0.001\ \text{MHz}$$

Note that we also receive the image frequency that "folds over" the primary output signals:

$$-f_c + f_{lo} = -14.001\ \text{MHz} + 14.000\ \text{MHz} = -0.001\ \text{MHz}$$

A low-pass filter easily removes the 28.001-MHz *sum frequency*, but the –0.001-MHz *difference-frequency* image will remain in the output. This unwanted image is the lower sideband with respect to the 14.000-MHz carrier frequency. This would not be a problem if there were no signals below 14.000 MHz to interfere. As previously stated, all undesired signals between 13.9985 and 14.000 MHz will translate into the passband along with the desired signals above 14.000 MHz. The image also results in increased noise in the output.

So how can we remove the image-frequency signals? It can be accomplished through *quadrature mixing*. Phasing or quadrature transmitters and receivers—also called Weaver-method or image-rejection mixers—have existed since the early days of single sideband. In fact, my first SSB transmitter was a used Central Electronics 20A exciter that incorporated a phasing design. Phasing systems lost favor in the early 1960s with the advent of relatively inexpensive, high-performance filters.

To achieve good opposite-sideband or image suppression, phasing systems require a precise balance of amplitude and phase between two samples of the signal that are 90° out

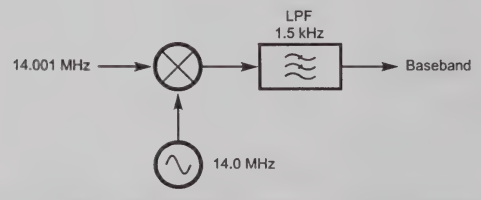


Fig 2—A direct-conversion real mixer with a 1.5-kHz low-pass filter.

Fig 3—Output spectrum of a real mixer illustrating the sum, difference and image frequencies.

of phase or in quadrature with each other—"orthogonal" is the term used in some texts. Until the advent of digital signal processing, it was difficult to realize the level of image rejection performance required of modern radio systems in phasing designs. Since digital signal processing allows precise numerical control of phase and amplitude, quadrature modulation and demodulation are the preferred methods. Such signals in quadrature allow virtually any modulation method to be implemented in software using DSP techniques.

Give Me *I* and *Q* and I Can Demodulate Anything

First, consider the direct-conversion mixer shown in Fig 2. When the RF signal is converted to baseband audio using a single channel, we can visualize the output as varying in amplitude along a single axis as illustrated in Fig 4. We will refer to this as the *in-phase* or *I* signal. Notice that its magnitude varies from a positive value to a negative value at the frequency of the modulating signal. If we use a diode to rectify the signal, we would have created a simple envelope or AM detector.

Remember that in AM envelope detection, both modulation sidebands carry information energy and both are desired at the output. Only amplitude information is required to fully demodulate the original signal. The problem is that most other modulation techniques require that the phase of the signal be known. This is where quadrature detection comes in. If we delay a copy of the RF carrier by 90° to form a quadrature (*Q*) signal, we can then use it in conjunction with the original in-phase signal and the math we learned in middle school to determine the instantaneous phase and amplitude of the original signal.

Fig 5 illustrates an RF carrier with the level of the *I* signal plotted on the x-axis and that of the *Q* signal plotted on the y-axis of a plane. This is often referred to in the literature as a *phasor diagram* in the *complex plane*. We are now able to extrapolate the two signals to draw an arrow or phasor that represents the instantaneous magnitude and phase of the original signal.

Okay, here is where you will have to use a couple of those extra functions on the calculator. To compute the magnitude m_t or envelope of the signal, we use the geometry of right triangles. In a right triangle, the square of the hypotenuse is equal to the sum of the squares of the other two sides—according to the Pythagorean theorem. Or restating, the hypotenuse as m_t (magnitude with respect to time):

$$m_t = \sqrt{I_t^2 + Q_t^2} \qquad \text{(Eq 3)}$$

The instantaneous phase of the signal as measured counterclockwise from the positive *I* axis and may be computed by the inverse tangent (or arctangent) as follows:

$$\phi_t = \tan^{-1}\left(\frac{Q_t}{I_t}\right) \qquad \text{(Eq 4)}$$

Therefore, if we measured the instantaneous values of *I* and *Q*, we would know everything we needed to know about the signal at a given moment in time. This is true whether we are dealing with continuous analog signals or discrete sampled signals. With *I* and *Q*, we can demodulate AM signals directly using Eq 3 and FM signals using Eq 4. To demodulate SSB takes one more step. Quadrature signals can be used analytically to remove the image frequencies and leave only the desired sideband.

The mathematical equations for quadrature signals are difficult but are very understandable with a little study.[2] I highly recommend that you read the online article, "Quadrature Signals: Complex, But Not Complicated," by Richard Lyons. It can be found at **www.dspguru.com/info/tutor/quadsig.htm**. The article develops in a very logical manner how quadrature-sampling *I/Q* demodulation is accomplished. A basic understanding of these concepts is essential to designing software-defined radios.

We can take advantage of the analytic capabilities of quadrature signals through a quadrature mixer. To understand the basic concepts of quadrature mixing, refer to Fig 6, which illustrates a quadrature-sampling *I/Q* mixer.

First, the RF input signal is bandpass filtered and applied to the two parallel mixer channels. By delaying the local oscillator wave by 90°, we can generate a cosine wave that, in tandem, forms a *quadrature oscillator*. The RF carrier, $f_c(t)$, is mixed with the respective cosine and sine wave local oscillators and is subsequently low-pass filtered to create the in-phase, $I(t)$, and quadrature, $Q(t)$, signals. The $Q(t)$

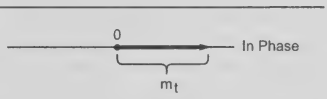


Fig 4—An in-phase signal (*I*) on the real plane. The magnitude, $m_{(t)}$, is easily measured as the instantaneous peak voltage, but no phase information is available from in-phase detection. This is the way an AM envelope detector works.

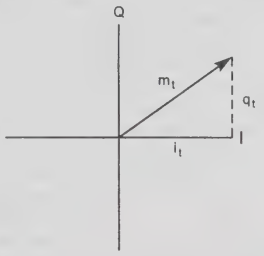


Fig 5—I +*j*Q are shown on the complex plane. The vector rotates counterclockwise at a rate of $2\pi f_c$. The magnitude and phase of the rotating vector at any instant in time may be determined through Eqs 3 and 4.

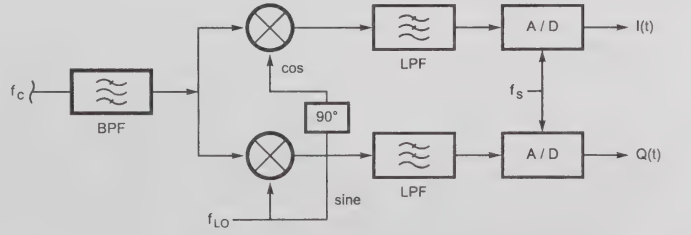


Fig 6—Quadrature sampling mixer: The RF carrier, f_c, is fed to parallel mixers. The local oscillator (Sine) is fed to the lower-channel mixer directly and is delayed by 90° (Cosine) to feed the upper-channel mixer. The low-pass filters provide antialias filtering before analog-to-digital conversion. The upper channel provides the in-phase ($I_{(t)}$) signal and the lower channel provides the quadrature ($Q_{(t)}$) signal. In the PC SDR the low-pass filters and A/D converters are integrated on the PC sound card.

channel is phase-shifted 90° relative to the $I(t)$ channel through mixing with the sine local oscillator. The low-pass filter is designed for cutoff below the Nyquist frequency to prevent aliasing in the A/D step. The A/D converts continuous-time signals to discrete-time sampled signals. Now that we have the I and Q samples in memory, we can perform the magic of digital signal processing.

Before we go further, let me reiterate that one of the problems with this method of down-conversion is that it can be costly to get good opposite-sideband suppression with analog circuits. Any variance in component values will cause phase or amplitude imbalance between two channels, resulting in a corresponding decrease in opposite-sideband suppression. With analog circuits, it is difficult to achieve better than 40 dB of suppression without much higher cost. Fortunately, it is straightforward to correct the analog imbalances in software.

Another significant drawback of direct-conversion receivers is that the noise increases as the demodulated signal approaches 0 Hz. Noise contributions come from a number of sources, such as $1/f$ noise from the semiconductor devices themselves, 60-Hz and 120-Hz line noise or hum, microphonic mechanical noise and local-oscillator phase noise near the carrier frequency. This can limit sensitivity since most people prefer their CW tones to be below 1 kHz. It turns out that most of the low-frequency noise rolls off above 1 kHz. Since a sound card can process signals all the way up to 20 kHz, why not use some of that bandwidth to move away from the low frequency noise? The PC SDR uses an 11.025-kHz, *offset-baseband IF* to reduce the noise to a manageable level. By offsetting the local oscillator by 11.025 kHz, we can now receive signals near the carrier frequency without any of the low-frequency noise issues. This also significantly reduces the effects of local-oscillator phase noise. Once we have digitally captured the signal, it is a trivial software task to shift the demodulated signal down to a 0-Hz offset.

DSP in the Frequency Domain

Every DSP text I have read thus far concentrates on time-domain filtering and demodulation of SSB signals using *finite-impulse-response (FIR)* filters. Since these techniques have been thoroughly discussed in the literature[1, 3, 4] and are not currently used in my PC SDR, they will not be covered in this article series.

My PC SDR uses the power of the *fast Fourier transform (FFT)* to do almost all of the heavy lifting in the frequency domain. Most DSP texts use a lot of ink to derive the math so that one can write the FFT code. Since Intel has so helpfully provided the code in executable form in their signal-processing library,[5] we don't care how to write an FFT: We just need to know how to use it. Simply put, the FFT converts the complex I and Q discrete-time signals into the frequency domain. The FFT output can be thought of as a large bank of very narrow band-pass filters, called *bins*, each one measuring the spectral energy within its respective bandwidth. The output resembles a *comb filter* wherein each bin slightly overlaps its adjacent bins forming a scalloped curve, as shown in Fig 7. When a signal is precisely at the center frequency of a bin, there will be a corresponding value only in that bin. As the frequency is offset from the bin's center, there will be a corresponding increase in the value of the adjacent bin and a decrease in the value of the current bin. Mathematical analysis fully describes the relationship between FFT bins,[6] but such is beyond the scope of this article.

Further, the FFT allows us to measure both phase and amplitude of the signal within each bin using Eqs 3 and 4 above. The complex version allows us to measure positive and negative frequencies separately. Fig 8 illustrates the output of a complex, or quadrature, FFT.

The bandwidth of each FFT bin may be computed as shown in Eq 5, where BW_{bin} is the bandwidth of a single bin, f_s is the sampling rate and N is the size of the FFT. The center frequency of each FFT bin may be determined by Eq 6 where f_{center} is the bin's center frequency, n is the bin number, f_s is the sampling rate and N is the size of the FFT. Bins zero through $(N/2)-1$ represent upper-sideband frequencies and bins $N/2$ to $N-1$ represent lower-sideband frequencies around the carrier frequency.

$$BW_{bin} = \frac{f_s}{N} \quad \text{(Eq 5)}$$

$$f_{center} = \frac{nf_s}{N} \quad \text{(Eq 6)}$$

If we assume the sampling rate of the sound card is 44.1 kHz and the number of FFT bins is 4096, then the bandwidth and center frequency of each bin would be:

$$BW_{bin} = \frac{44100}{4096} = 10.7666 \text{ Hz and}$$

$$f_{center} = n10.7666 \text{ Hz}$$

What this all means is that the receiver will have 4096, ~11-Hz-wide

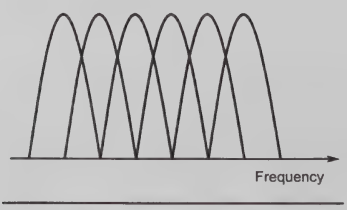


Fig 7—FFT output resembles a comb filter: Each bin of the FFT overlaps its adjacent bins just as in a comb filter. The 3-dB points overlap to provide linear output. The phase and magnitude of the signal in each bin is easily determined mathematically with Eqs 3 and 4.

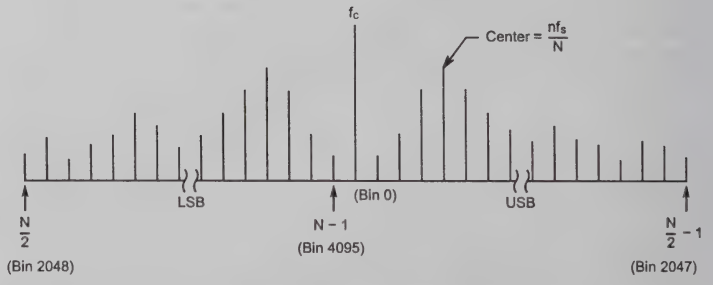


Fig 8—Complex FFT output: The output of a complex FFT may be thought of as a series of band-pass filters aligned around the carrier frequency, f_c, at bin 0. N represents the number of FFT bins. The upper sideband is located in bins 1 through $(N/2)-1$ and the lower sideband is located in bins $N/2$ to $N-1$. The center frequency and bandwidth of each bin may be calculated using Eqs 5 and 6.

band-pass filters. We can therefore create band-pass filters from 11 Hz to approximately 40 kHz in 11-Hz steps.

The PC SDR performs the following functions in the frequency domain after FFT conversion:

- Brick-wall fixed and variable band-pass filters
- Frequency conversion
- SSB/CW demodulation
- Sideband selection
- Frequency-domain noise subtraction
- Frequency-selective squelch
- Noise blanking
- Graphic equalization ("tone control")
- Phase and amplitude balancing to remove images
- SSB generation
- Future digital modes such as PSK31 and RTTY

Once the desired frequency-domain processing is completed, it is simple to convert the signal back to the time domain by using an *inverse FFT*. In the PC SDR, only AGC and adaptive noise filtering are currently performed in the time domain. A simplified diagram of the PC SDR software architecture is provided in Fig 9. These concepts will be discussed in detail in a future article.

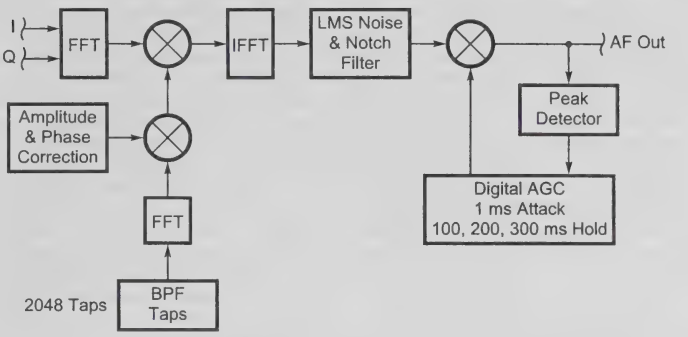


Fig 9—SDR receiver software architecture: The *I* and *Q* signals are fed from the sound-card input directly to a 4096-bin complex FFT. Band-pass filter coefficients are precomputed and converted to the frequency domain using another FFT. The frequency-domain filter is then multiplied by the frequency-domain signal to provide brick-wall filtering. The filtered signal is then converted to the time domain using the inverse FFT. Adaptive noise and notch filtering and digital AGC follow in the time domain.

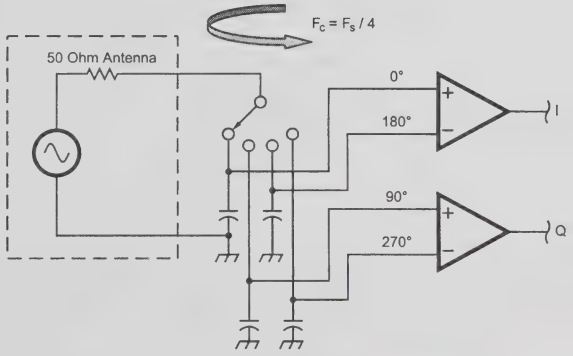


Fig 10—Tayloe detector: The switch rotates at the carrier frequency so that each capacitor samples the signal once each revolution. The 0° and 180° capacitors differentially sum to provide the in-phase (*I*) signal and the 90° and 270° capacitors sum to provide the quadrature (*Q*) signal.

Sampling RF Signals with the Tayloe Detector: A New Twist on an Old Problem

While searching the Internet for information on quadrature mixing, I ran across a most innovative and elegant design by Dan Tayloe, N7VE. Dan, who works for Motorola, has developed and patented (US Patent #6,230,000) what has been called the *Tayloe detector*[7]. The beauty of the Tayloe detector is found in both its design elegance and its exceptional performance. It resembles other concepts in design, but appears unique in its high performance with minimal components.[8, 9, 10, 11] In its simplest form, you can build a complete quadrature down converter with only three or four ICs (less the local oscillator) at a cost of less than $10.

Fig 10 illustrates a single-balanced version of the Tayloe detector. It can be visualized as a four-position rotary switch revolving at a rate equal to the carrier frequency. The 50-Ω antenna impedance is connected to the rotor and each of the four switch positions is connected to a *sampling capacitor*. Since the switch rotor is turning at exactly the RF carrier frequency, each capacitor will track the carrier's amplitude for exactly one-quarter of the cycle and will hold its value for the remainder of the cycle. The rotating switch will therefore sample the signal at 0°, 90°, 180° and 270°, respectively.

As shown in Fig 11, the 50-Ω impedance of the antenna and the sampling capacitors form an R-C low-pass filter during the period when each respective switch is turned on. Therefore, each sample represents the integral or average voltage of the signal during its respective one-quarter cycle. When the switch is off, each sampling capacitor will hold its value until the next revolution. If the RF carrier and the rotating frequency were exactly in phase, the output of each capacitor will be a dc level equal to the average

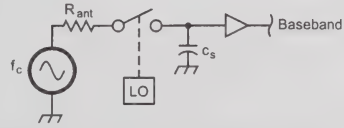


Fig 11—Track and hold sampling circuit: Each of the four sampling capacitors in the Tayloe detector form an RC track-and-hold circuit. When the switch is on, the capacitor will charge to the average value of the carrier during its respective one-quarter cycle. During the remaining three-quarters cycle, it will hold its charge. The local-oscillator frequency is equal to the carrier frequency so that the output will be at baseband.

value of the sample.

If we differentially sum outputs of the 0° and 180° sampling capacitors with an op amp (see Fig 10), the output would be a dc voltage equal to two times the value of the individually sampled values when the switch rotation frequency equals the carrier frequency. Imagine, 6 dB of noise-free gain! The same would be true for the 90° and 270° capacitors as well. The 0°/180° summation forms the *I* channel and the 90°/270° summation forms the *Q* channel of the quadrature down-conversion.

As we shift the frequency of the carrier away from the sampling frequency, the values of the inverting phases will no longer be dc levels. The output frequency will vary according to the "beat" or difference frequency between the carrier and the switch-rotation frequency to provide an accurate representation of all the signal components converted to baseband.

Fig 12 provides the schematic for a simple, single-balanced Tayloe detector. It consists of a PI5V331, 1:4 FET demultiplexer that switches the signal to each of the four sampling capacitors. The 74AC74 dual flip-flop is connected as a divide-by-four Johnson counter to provide the two-phase clock to the demultiplexer chip. The outputs of the sampling capacitors are differentially summed through the two LT1115 ultra-low-noise op amps to form the *I* and *Q* outputs, respectively. Note that the impedance of the antenna forms the input resistance for the op-amp gain as shown in Eq 7. This impedance may vary significantly with the actual antenna. I use instrumentation amplifiers in my final design to eliminate gain variance with antenna impedance. More information on the hardware design will be provided in a future article.

Since the duty cycle of each switch is 25%, the effective resistance in the RC network is the antenna impedance multiplied by four in the op-amp gain formula, as shown in Eq 7:

$$G = \frac{R_f}{4R_{ant}} \qquad \text{(Eq 7)}$$

For example, with a feedback resistance, R_f, of 3.3 kΩ and antenna impedance, R_{ant}, of 50 Ω, the resulting gain of the input stage is:

$$G = \frac{3300}{4 \times 50} = 16.5$$

The Tayloe detector may also be analyzed as a *digital commutating filter*.[12, 13, 14] This means that it operates as a very-high-*Q* tracking filter, where Eq 8 determines the bandwidth and *n* is the number of sampling capacitors,

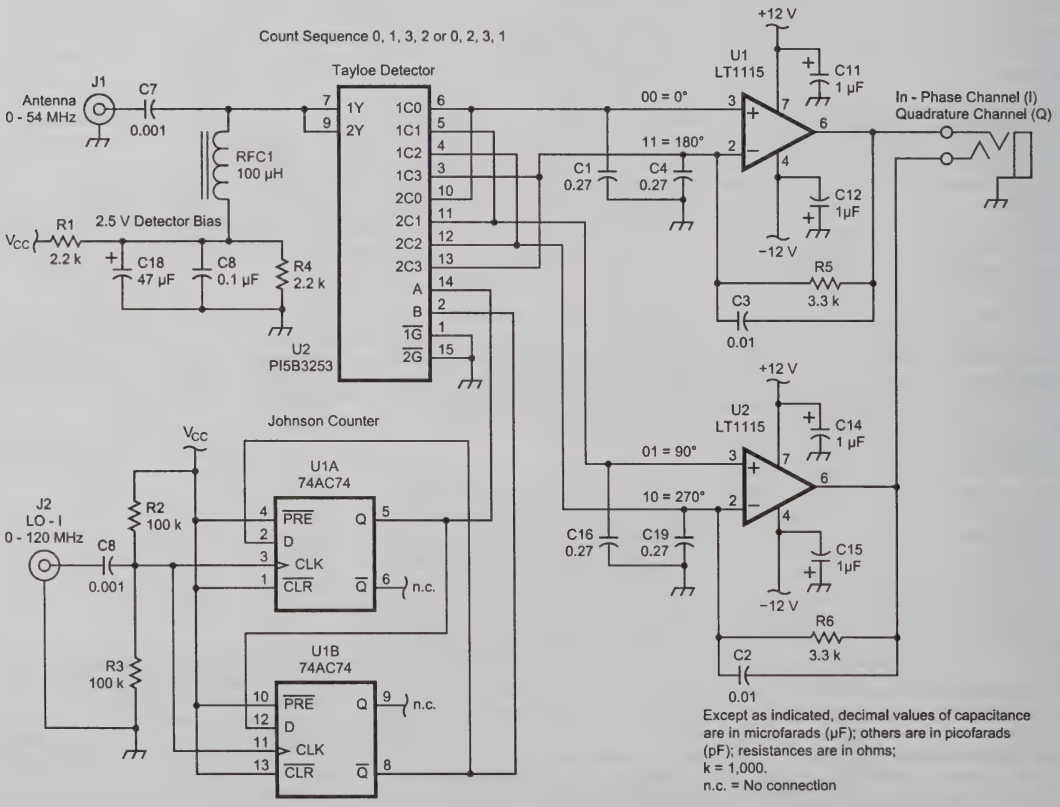


Fig 12—Singly balanced Tayloe detector.

R_{ant} is the antenna impedance and C_s is the value of the individual sampling capacitors. Eq 9 determines the Q_{det} of the filter, where f_c is the center frequency and BW_{det} is the bandwidth of the filter.

$$BW_{det} = \frac{1}{\pi n R_{ant} C_s} \quad \text{(Eq 8)}$$

$$Q_{det} = \frac{f_c}{BW_{det}} \quad \text{(Eq 9)}$$

By example, if we assume the sampling capacitor to be 0.27 µF and the antenna impedance to be 50 Ω, then BW and Q are computed as follows:

$$BW_{det} = \frac{1}{(\pi)(4)(50)(2.7\times10^{-7})} = 5895 \text{ Hz}$$

$$Q_{det} = \frac{14.001\times10^{6}}{5895} = 2375$$

Since the PC SDR uses an offset baseband IF, I have chosen to design the detector's bandwidth to be 40 kHz to allow low-frequency noise elimination as discussed above.

The real payoff in the Tayloe detector is its performance. It has been stated that the *ideal* commutating mixer has a minimum conversion loss (which equates to noise figure) of 3.9 dB.[15, 16] Typical high-level diode mixers have a conversion loss of 6-7 dB and noise figures 1 dB higher than the loss. The Tayloe detector has less than 1 dB of conversion loss, remarkably. How can this be? The reason is that it is not really a mixer but a sampling detector in the form of a quadrature track and hold. This means that the design adheres to discrete-time sampling theory, which, while similar to mixing, has its own unique characteristics. Because a track and hold actually holds the signal value between samples, the signal output never goes to zero.

This is where aliasing can actually be used to our benefit. Since each switch and capacitor in the Tayloe detector actually samples the RF signal once each cycle, it will respond to alias frequencies as well as those within the Nyquist frequency range. In a traditional direct-conversion receiver, the local-oscillator frequency is set to the carrier frequency so that the difference frequency, or IF, is at 0 Hz and the sum frequency is at two times the carrier frequency per Eq 2. We normally remove the sum frequency through low-pass filtering, resulting in conversion loss and a corresponding increase in noise figure. In the Tayloe detector, the sum frequency resides at the first alias frequency as shown in Fig 13. Remember that an alias is a real signal and will appear in the output as if it were a baseband signal. Therefore, the alias adds to the baseband signal for a theoretically lossless detector. In real life, there is a slight loss due to the resistance of the switch and aperture loss due to imperfect switching times.

PC SDR Transceiver Hardware

The Tayloe detector therefore provides a low-cost, high-performance method for both quadrature down-conversion as well as up-conversion for transmitting. For a complete system, we would need to provide analog AGC to prevent overload of the ADC inputs and a means of digital frequency control. Fig 14 illustrates the hardware architecture of the PC SDR receiver as it currently exists. The challenge has been to build a low-noise analog chain that matches the dynamic range of the Tayloe detector to the dynamic range of the PC sound card. This will be covered in a future article.

I am currently prototyping a complete PC SDR transceiver, the SDR-1000, that will provide general-coverage receive from 100 kHz to 54 MHz and will transmit on all ham bands from 160 through 6 meters.

SDR Applications

At the time of this writing, the typical entry-level PC now runs at a clock frequency greater than 1 GHz and costs only a few hundred dollars. We now have exceptional processing power at our disposal to perform DSP tasks that were once only dreams. The transfer of knowledge from the aca-

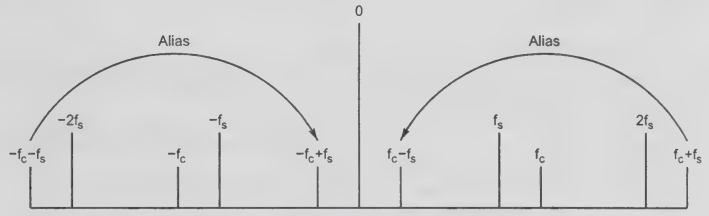


Fig 13—Alias summing on Tayloe detector output: Since the Tayloe detector samples the signal the sum frequency (*f*c + *f*s) and its image (–*f*c – *f*s) are located at the first alias frequency. The alias signals sum with the baseband signals to eliminate the mixing product loss associated with traditional mixers. In a typical mixer, the sum frequency energy is lost through filtering thereby increasing the noise figure of the device.

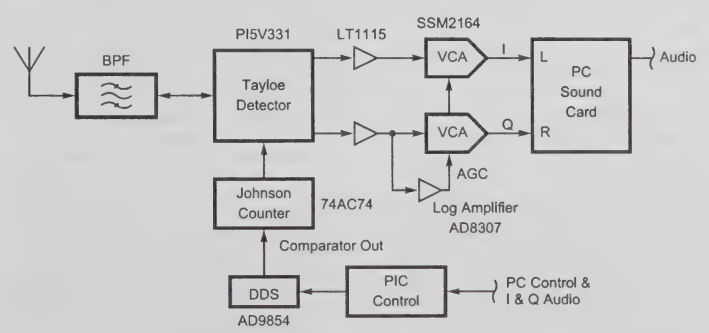


Fig 14—PC SDR receiver hardware architecture: After band-pass filtering the antenna is fed directly to the Tayloe detector, which in turn provides *I* and *Q* outputs at baseband. A DDS and a divide-by-four Johnson counter drive the Tayloe detector demultiplexer. The LT1115s offer ultra-low noise-differential summing and amplification prior to the wide-dynamic-range analog AGC circuit formed by the SSM2164 and AD8307 log amplifier.

demic to the practical is the primary limit of the availability of this technology to the Amateur Radio experimenter. This article series attempts to demystify some of the fundamental concepts to encourage experimentation within our community. The ARRL recently formed a SDR Working Group for supporting this effort, as well.

The SDR mimics the analog world in digital data, which can be manipulated much more precisely. Analog radio has always been modeled mathematically and can therefore be processed in a computer. This means that virtually any modulation scheme may be handled digitally with performance levels difficult, or impossible, to attain with analog circuits. Let's consider some of the amateur applications for the SDR:

- Competition-grade HF transceivers
- High-performance IF for microwave bands
- Multimode digital transceiver
- EME and weak-signal work
- Digital-voice modes
- Dream it and code it

For Further Reading

For more in-depth study of DSP techniques, I highly recommend that you purchase the following texts in order of their listing:

Understanding Digital Signal Processing by Richard G. Lyons (see Note 6). This is one of the best-written textbooks about DSP.

Digital Signal Processing Technology by Doug Smith (see Note 4). This new book explains DSP theory and application from an Amateur Radio perspective.

Digital Signal Processing in Communications Systems by Marvin E. Frerking (see Note 3). This book relates DSP theory specifically to modulation and demodulation techniques for radio applications.

Acknowledgements

I would like to thank those who have assisted me in my journey to understanding software radios. Dan Tayloe, N7VE, has always been helpful and responsive in answering questions about the Tayloe detector. Doug Smith, KF6DX, and Leif Åsbrink, SM5BSZ, have been gracious to answer my questions about DSP and receiver design on numerous occasions. Most of all, I want to thank my Saturday-morning breakfast review team: Mike Pendley, WA5VTV; Ken Simmons, K5UHF; Rick Kirchhof, KD5ABM; and Chuck McLeavy, WB5BMH. These guys put up with my questions every week and have given me tremendous advice and feedback all throughout the project. I also want to thank my wonderful wife, Virginia, who has been incredibly patient with all the hours I have put in on this project.

Where Do We Go From Here?

Three future articles will describe the construction and programming of the PC SDR. The next article in the series will detail the software interface to the PC sound card. Integrating full-duplex sound with *DirectX* was one of the more challenging parts of the project. The third article will describe the *Visual Basic* code and the use of the Intel Signal Processing Library for implementing the key DSP algorithms in radio communications. The final article will describe the completed transceiver hardware for the SDR-1000.

Notes

[1]D. Smith, KF6DX, "Signals, Samples and Stuff: A DSP Tutorial (Part 1)," *QEX,* Mar/Apr 1998, pp 3-11.

[2]J. Bloom, KE3Z, "Negative Frequencies and Complex Signals," *QEX*, Sep 1994, pp 22-27.

[3]M. E. Frerking, *Digital Signal Processing in Communication Systems* (New York: Van Nostrand Reinhold, 1994, ISBN: 0442016166), pp 272-286.

[4]D. Smith, KF6DX, *Digital Signal Processing Technology* (Newington, Connecticut: ARRL, 2001), pp 5-1 through 5-38.

[5]The Intel Signal Processing Library is available for download at **developer.intel.com/software/products/perflib/spl/**.

[6]R. G. Lyons, *Understanding Digital Signal Processing*, (Reading, Massachusetts: Addison-Wesley, 1997), pp 49-146.

[7]D. Tayloe, N7VE, "Letters to the Editor, Notes on'Ideal' Commutating Mixers (Nov/Dec 1999)," *QEX,* March/April 2001, p 61.

[8]P. Rice, VK3BHR, "SSB by the Fourth Method?" available at **ironbark.bendigo.latrobe.edu.au/~rice/ssb/ssb.html**.

[9]A. A. Abidi, "Direct-Conversion Radio Transceivers for Digital Communications," *IEEE Journal of Solid-State Circuits*, Vol 30, No 12, December 1995, pp 1399-1410, Also on the Web at **www.icsl.ucla.edu/aagroup/PDF_files/dir-con.pdf**

[10]P. Y. Chan, A. Rofougaran, K.A. Ahmed, and A. A. Abidi, "A Highly Linear 1-GHz CMOS Downconversion Mixer." Presented at the European Solid State Circuits Conference, Seville, Spain, Sep 22-24, 1993, pp 210-213 of the conference proceedings. Also on the Web at **www.icsl.ucla.edu/aagroup/PDF_files/mxr-93.pdf**

[11]D. H. van Graas, PA0DEN, "The Fourth Method: Generating and Detecting SSB Signals," *QEX*, Sep 1990, pp 7-11. This circuit is very similar to a Tayloe detector, but it has a lot of unnecessary components.

[12]M. Kossor, WA2EBY, "A Digital Commutating Filter," *QEX,* May/Jun 1999, pp 3-8.

[13]C. Ping, BA1HAM, "An Improved Switched Capacitor Filter," *QEX,* Sep/Oct 2000, pp 41-45.

[14]P. Anderson, KC1HR, "Letters to the Editor, A Digital Commutating Filter," *QEX,* Jul/Aug 1999, pp 62.

[15]D. Smith, KF6DX, "Notes on 'Ideal' Commutating Mixers," *QEX,* Nov/Dec 1999, pp 52-54.

[16]P. Chadwick, G3RZP, "Letters to the Editor, Notes on 'Ideal' Commutating Mixers" (Nov/Dec 1999), *QEX,* Mar/Apr 2000, pp 61-62.

Gerald became a ham in 1967 during high school, first as a Novice and then a General class as WA5RXV. He completed his Advanced class license and became KE5OH before finishing high school and received his First Class Radiotelephone license while working in the television broadcast industry during college. After 25 years of inactivity, Gerald returned to the active amateur ranks in 1997 when he completed the requirements for Extra class license and became AC5OG.

Gerald lives in Austin, Texas, and is currently CEO of Sixth Market Inc, a hedge fund that trades equities using artificial-intelligence software. Gerald previously founded and ran five technology companies spanning hardware, software and electronic manufacturing. Gerald holds a Bachelor of Science Degree in Electrical Engineering from Mississippi Stage University.

Gerald is a member of the ARRL SDR working Group and currently enjoys homebrew software-radio development, 6-meter DX and satellite operations. □□

A Software-Defined Radio for the Masses, Part 2

Come learn how to use a PC sound card to enter the wonderful world of digital signal processing.

By Gerald Youngblood, AC5OG

Part 1 gave a general description of digital signal processing (DSP) in software-defined radios (SDRs).[1] It also provided an overview of a full-featured radio that uses a personal computer to perform all DSP functions. This article begins design implementation with a complete description of software that provides a full-duplex interface to a standard PC sound card.

To perform the magic of digital signal processing, we must be able to convert a signal from analog to digital and back to analog again. Most amateur experimenters already have this capability in their shacks and many have used it for slow-scan television or the new digital modes like PSK31.

Part 1 discussed the power of quadrature signal processing using *in-phase* (I) and *quadrature* (Q) signals to receive or transmit using virtually any modulation method. Fortunately, all modern PC sound cards offer the perfect method for digitizing the I and Q signals. Since virtually all cards today provide 16-bit stereo at 44-kHz sampling rates, we have exactly what we need capture and process the signals in software. Fig 1 illustrates a direct quadrature-conversion mixer connection to a PC sound card.

This article discusses complete source code for a DirectX sound-card interface in Microsoft *Visual Basic*. Consequently, the discussion assumes that the reader has some fundamental knowledge of high-level language programming.

[1]Notes appear on page 18.

8900 Marybank Dr
Austin, TX 78750
gerald@sixthmarket.com

Sound Card and PC Capabilities

Very early PC sound cards were low-performance, 8-bit mono versions. Today, virtually all PCs come with 16-bit stereo cards of sufficient quality to be used in a software-defined radio. Such a card will allow us to demodulate, filter and display up to approximately a 44-kHz bandwidth, assuming a 44-kHz sampling rate. (The bandwidth is 44 kHz, rather than 22 kHz, because the use of two channels effectively doubles the sampling rate—*Ed.*) For high-performance applications, it is important to select a card that offers a high dynamic range—on the order of 90 dB. If you are just getting started, most PC sound cards will allow you to begin experimentation, although they

may offer lower performance.

The best 16-bit price-to-performance ratio I have found at the time of this article is the Santa Cruz 6-channel DSP Audio Accelerator from Turtle Beach Inc (**www.tbeach.com**). It offers four 18-bit internal analog-to-digital (A/D) input channels and six 20-bit digital-to-analog (D/A) output channels with sampling rates up to 48 kHz. The manufacturer specifies a 96-dB signal-to-noise ratio (SNR) and better than –91 dB total harmonic distortion plus noise (THD+N). Crosstalk is stated to be –105 dB at 100 Hz. The Santa Cruz card can be purchased from online retailers for under $70.

Each bit on an A/D or D/A converter represents 6 dB of dynamic range, so a 16-bit converter has a theoretical limit of 96 dB. A very good converter with low-noise design is required to achieve this level of performance. Many 16-bit sound cards provide no more than 12-14 effective bits of dynamic range. To help achieve higher performance, the Santa Cruz card uses an 18-bit A/D converter to deliver the 96 dB dynamic range (16-bit) specification.

A SoundBlaster 64 also provides reasonable performance on the order of 76 dB SNR according to PC AV Tech at **www.pcavtech.com**. I have used this card with good results, but I much prefer the Santa Cruz card.

The processing power needed from the PC depends greatly on the signal processing required by the application. Since I am using very-high-performance filters and large fast-Fourier transforms (FFTs), my applications require at least a 400-MHz Pentium II processor with a minimum of 128 MB of RAM. If you require less performance from the software, you can get by with a much slower machine. Since the entry level for new PCs is now 1 GHz, many amateurs have ample processing power available.

Microsoft DirectX versus Windows Multimedia

Digital signal processing using a PC sound card requires that we be able to capture blocks of digitized *I* and *Q* data through the stereo inputs, process those signals and return them to the sound-card outputs in pseudo real time. This is called *full duplex*. Unfortunately, there is no high-level software interface that offers the capabilities we need for the SDR application.

Microsoft now provides two application programming interfaces[2] (APIs) that allow direct access to the sound card under *C++* and *Visual Basic*. The original interface is the Windows Multimedia system using the Waveform Audio API. While my early work was done with the Waveform Audio API, I later abandoned it for the higher performance and simpler interface DirectX offers. The only limitation I have found with DirectX is that it does not currently support sound cards with more than 16-bits of resolution. For 24-bit cards, Windows Multimedia is required. While the Santa Cruz card supports 18-bits internally, it presents only 16-bits to the interface. For information on where to download the DirectX software development kit (SDK) see Note 2.

Circular Buffer Concepts

A typical full-duplex PC sound card allows the simultaneous capture and playback of two or more audio channels (stereo). Unfortunately, there is no high-level code in *Visual Basic* or C++ to directly support full duplex as required in an SDR. We will therefore have to write code to directly control the card through the DirectX API.

DirectX internally manages all low-level buffers and their respective interfaces to the sound-card hardware. Our code will have to manage the high-level DirectX buffers (called DirectSoundBuffer and DirectSoundCaptureBuffer) to provide uninterrupted operation in a multitasking system. The DirectSoundCaptureBuffer stores the digitized signals from the stereo

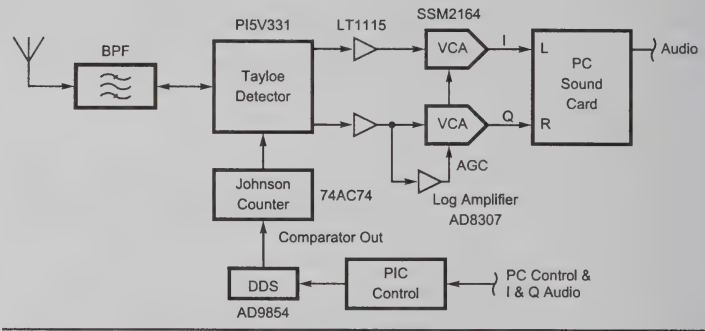


Fig 1—Direct quadrature conversion mixer to sound-card interface used in the author's prototype.

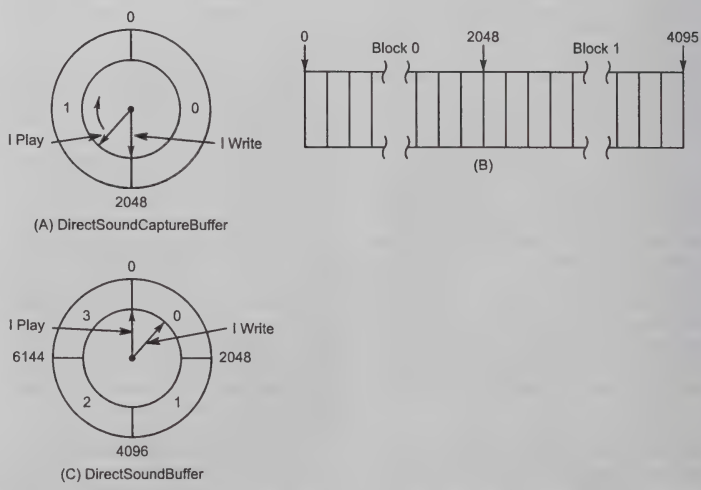


Fig 2—DirectSoundCaptureBuffer and DirectSoundBuffer circular buffer layout.

A/D converter in a circular buffer and notifies the application upon the occurrence of predefined events. Once captured in the buffer, we can read the data, perform the necessary modulation or demodulation functions using DSP and send the data to the DirectSoundBuffer for D/A conversion and output to the speakers or transmitter.

To provide smooth operation in a multitracking system without audio popping or interruption, it will be necessary to provide a multilevel buffer for both capture and playback. You may have heard the term double buffering. We will use double buffering in the DirectSoundCaptureBuffer and quadruple buffering in the DirectSoundBuffer. I found that the quad buffer with overwrite detection was required on the output to prevent overwriting problems when the system is heavily loaded with other applications. Figs 2A and 2B illustrate the concept of a circular double buffer, which is used for the DirectSoundCaptureBuffer. Although the buffer is really a linear array in memory, as shown in Fig 2B, we can visualize it as circular, as illustrated in Fig 2A. This is so because DirectX manages the buffer so that as soon as each cursor reaches the end of the array, the driver resets the cursor to the beginning of the buffer.

The DirectSoundCaptureBuffer is broken into two blocks, each equal in size to the amount of data to be captured and processed between each event. Note that an event is much like an interrupt. In our case, we will use a block size of 2048 samples. Since we are using a stereo (two-channel) board with 16 bits per channel, we will be capturing 8192 bytes per block (2048 samples × 2 channels × 2 bytes). Therefore, the DirectSoundCaptureBuffer will be twice as large (16,384 bytes).

Since the DirectSoundCapture Buffer is divided into two data blocks, we will need to send an event notification to the application after each block has been captured. The DirectX driver maintains cursors that track the position of the capture operation at all times. The driver provides the means of setting specific locations within the buffer that cause an event to trigger, thereby telling the application to retrieve the data. We may then read the correct block directly from the DirectSoundCaptureBuffer segment that has been completed.

Referring again to Fig 2A, the two cursors resemble the hands on a clock face rotating in a clockwise direction. The capture cursor, lPlay, represents the point at which data are currently being captured. (I know that sounds backward, but that is how Microsoft defined it.) The read cursor, lWrite, trails the capture cursor and indicates the point up to which data can safely be read. The data after lWrite and up to and including lPlay are not necessarily good data because of hardware buffering. We can use the lWrite cursor to trigger an event that tells the software to read each respective block of data, as will be discussed later in the article. We will therefore receive two events per revolution of the circular buffer. Data can be captured into one half of the buffer while data are being read from the other half.

Fig 2C illustrates the DirectSoundBuffer, which is used to output data to the D/A converters. In this case, we will use a quadruple buffer to allow plenty of room between the currently playing segment and the segment being written. The play cursor, lPlay, always points to the next byte of data to be played. The write cursor, lWrite, is the point after which it is safe to write data into the buffer. The cursors may be thought of as rotating in a clockwise motion just as the capture cursors do. We must monitor the location of the cursors before writing to buffer locations between the cursors to prevent overwriting data that have already been committed to the hardware for playback.

Now let's consider how the data maps from the DirectSoundCaptureBuffer to the DirectSoundBuffer. To prevent gaps or pops in the sound due to processor loading, we will want to fill the entire quadruple buffer before starting the playback looping. DirectX allows the application to set the starting point for the lPlay cursor and to start the playback at any time. Fig 3 shows how the data blocks map sequentially from the DirectSoundCaptureBuffer to the DirectSoundBuffer. Block 0 from the DirectSoundCaptureBuffer is transferred to Block 0 of the DirectSoundBuffer. Block 1 of the DirectSoundCaptureBuffer is next transferred to Block 1 of the DirectSoundBuffer and so forth. The subsequent source-code examples show how control of the buffers is accomplished.

Full Duplex, Step-by-Step

The following sections provide a detailed discussion of full-duplex DirectX implementation. The example code captures and plays back a stereo audio signal that is delayed by four

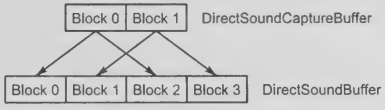


Fig 3—Method for mapping the DirectSoundCaptureBuffer to the DirectSoundBuffer.

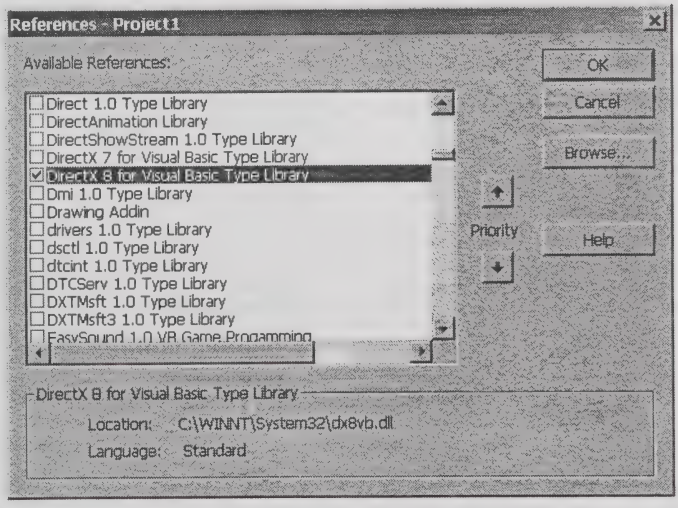


Fig 4—Registration of the DirectX8 for *Visual Basic* Type Library in the *Visual Basic* IDE.

capture periods through buffering. You should refer to the "DirectX Audio" section of the *DirectX 8.0 Programmers Reference* that is installed with the DirectX software developer's kit (SDK) throughout this discussion. The DSP code will be discussed in the next article of this series, which will discuss the modulation and demodulation of quadrature signals in the SDR. Here are the steps involved in creating the DirectX interface:

- Install DirectX runtime and SDK.
- Add a reference to DirectX8 for *Visual Basic* Type Library.
- Define Variables, I/O buffers and DirectX objects.
- Implement DirectX8 events and event handles.
- Create the audio devices.
- Create the DirectX events.
- Start and stop capture and play buffers.
- Process the DirectXEvent8.
- Fill the play buffer before starting playback.
- Detect and correct overwrite errors.
- Parse the stereo buffer into *I* and *Q* signals.
- Destroy objects and events on exit.

Complete functional source code for the DirectX driver written in Microsoft *Visual Basic* is provided for download from the *QEX* Web site.[3]

Install DirectX and Register it within Visual Basic

The first step is to download the DirectX driver and the DirectX SDK

```
Option Explicit

'Define Constants
Const Fs As Long = 44100          'Sampling frequency Hz
Const NFFT As Long = 4096         'Number of FFT bins
Const BLKSIZE As Long = 2048      'Capture/play block size
Const CAPTURESIZE As Long = 4096       'Capture Buffer size

'Define DirectX Objects
Dim dx As New DirectX8            'DirectX object
Dim ds As DirectSound8            'DirectSound object
Dim dspb As DirectSoundPrimaryBuffer8 'Primary buffer object
Dim dsc As DirectSoundCapture8    'Capture object
Dim dsb As DirectSoundSecondaryBuffer8 'Output Buffer object
Dim dscb As DirectSoundCaptureBuffer8 'Capture Buffer object

'Define Type Definitions
Dim dscbd As DSCBUFFERDESC        'Capture buffer description
Dim dsbd As DSBUFFERDESC          'DirectSound buffer description
Dim dspbd As WAVEFORMATEX         'Primary buffer description
Dim CapCurs As DSCURSORS          'DirectSound Capture Cursor
Dim PlyCurs As DSCURSORS          'DirectSound Play Cursor

'Create I/O Sound Buffers
Dim inBuffer(CAPTURESIZE) As Integer   'Demodulator Input Buffer
Dim outBuffer(CAPTURESIZE) As Integer 'Demodulator Output Buffer

'Define pointers and counters
Dim Pass As Long                  'Number of capture passes
Dim InPtr As Long                 'Capture Buffer block pointer
Dim OutPtr As Long                'Output Buffer block pointer
Dim StartAddr As Long             'Buffer block starting address
Dim EndAddr As Long               'Ending buffer block address
Dim CaptureBytes As Long          'Capture bytes to read

'Define loop counter variables for timing the capture event cycle
Dim TimeStart As Double           'Start time for DirectX8Event loop
Dim TimeEnd As Double             'Ending time for DirectX8Event loop
Dim AvgCtr As Long                'Counts number of events to average
Dim AvgTime As Double             'Stores the average event cycle time

'Set up Event variables for the Capture Buffer
Implements DirectXEvent8          'Allows DirectX Events
Dim hEvent(1) As Long             'Handle for DirectX Event
Dim EVNT(1) As DSBPOSITIONNOTIFY       'Notify position array
Dim Receiving As Boolean          'In Receive mode if true
Dim FirstPass As Boolean          'Denotes first pass from Start
```

Fig 5—Declaration of variables, buffers, events and objects. This code is located in the General section of the module or form.

from the Microsoft Web site (see Note 3). Once the driver and SDK are installed, you will need to register the DirectX8 for *Visual Basic* Type Library within the *Visual Basic* development environment.

If you are building the project from scratch, first create a *Visual Basic* project and name it "Sound." When the project loads, go to the Project Menu/References, which loads the form shown in Fig 4. Scroll through Available References until you locate the DirectX8 for *Visual Basic* Type Library and check the box. When you press "OK," the library is registered.

Define Variables, Buffers and DirectX Objects

Name the form in the Sound project frmSound. In the General section of frmSound, you will need to declare all of the variables, buffers and DirectX objects that will be used in the driver interface. Fig 5 provides the code that is to be copied into the General section. All definitions are commented in the code and should be self-explanatory when viewed in conjunction with the subroutine code.

Create the Audio Devices

We are now ready to create the DirectSound objects and set up the format of the capture and play buffers. Refer to the source code in Fig 6 during the following discussion.

The first step is to create the DirectSound and DirectSoundCapture

```
'Set up the DirectSound Objects and the Capture and Play Buffers
Sub CreateDevices()

    On Local Error Resume Next

    Set ds = dx.DirectSoundCreate(vbNullString)     'DirectSound object
    Set dsc = dx.DirectSoundCaptureCreate(vbNullString)   'DirectSound Capture

    'Check to se if Sound Card is properly installed
    If Err.Number <> 0 Then
        MsgBox "Unable to start DirectSound. Check proper sound card installation"
        End
    End If

    'Set the cooperative level to allow the Primary Buffer format to be set
    ds.SetCooperativeLevel Me.hWnd, DSSCL_PRIORITY

    'Set up format for capture buffer
    With dscbd
        With .fxFormat
            .nFormatTag = WAVE_FORMAT_PCM
            .nChannels = 2                  'Stereo
            .lSamplesPerSec = Fs            'Sampling rate in Hz
            .nBitsPerSample = 16            '16 bit samples
            .nBlockAlign = .nBitsPerSample / 8 * .nChannels
            .lAvgBytesPerSec = .lSamplesPerSec * .nBlockAlign
        End With
        .lFlags = DSCBCAPS_DEFAULT
        .lBufferBytes = (dscbd.fxFormat.nBlockAlign * CAPTURESIZE) 'Buffer Size
        CaptureBytes = .lBufferBytes \ 2     'Bytes for 1/2 of capture buffer
    End With

    Set dscb = dsc.CreateCaptureBuffer(dscbd)      'Create the capture buffer

    ' Set up format for secondary playback buffer
    With dsbd
        .fxFormat = dscbd.fxFormat
        .lBufferBytes = dscbd.lBufferBytes * 2     'Play is 2X Capture Buffer Size
        .lFlags = DSBCAPS_GLOBALFOCUS Or DSBCAPS_GETCURRENTPOSITION2
    End With

    dspbd = dsbd.fxFormat                   'Set Primary Buffer format
    dspb.SetFormat dspbd                    'to same as Secondary Buffer

    Set dsb = ds.CreateSoundBuffer(dsbd)      'Create the secondary buffer

End Sub
```

Fig 6—Create the DirectX capture and playback devices.

objects. We then check for an error to see if we have a compatible sound card installed. If not, an error message would be displayed to the user. Next, we set the cooperative level DSSCL_ PRIORITY to allow the Primary Buffer format to be set to the same as that of the Secondary Buffer. The code that follows sets up the DirectSoundCaptureBufferDescription format and creates the DirectSoundCaptureBuffer object. The format is set to 16-bit stereo at the sampling rate set by the constant *Fs*.

Next, the DirectSoundBufferDescription is set to the same format as the DirectSoundCaptureBufferDescription. We then set the Primary Buffer format to that of the Secondary Buffer before creating the DirectSoundBuffer object.

Set the DirectX Events

As discussed earlier, the DirectSoundCaptureBuffer is divided into two blocks so that we can read from one block while capturing to the other. To do so, we must know when

```
'Set events for capture buffer notification at 0 and 1/2
Sub SetEvents()

    hEvent(0) = dx.CreateEvent(Me)          'Event handle for first half of buffer
    hEvent(1) = dx.CreateEvent(Me)          'Event handle for second half of buffer

    'Buffer Event 0 sets Write at 50% of buffer
    EVNT(0).hEventNotify = hEvent(0)
    EVNT(0).lOffset = (dscbd.lBufferBytes \ 2) - 1'Set event to first half of capture buffer

    'Buffer Event 1 Write at 100% of buffer
    EVNT(1).hEventNotify = hEvent(1)
    EVNT(1).lOffset = dscbd.lBufferBytes - 1      'Set Event to second half of capture buffer

    dscb.SetNotificationPositions 2, EVNT()       'Set number of notification positions to 2

End Sub
```

Fig 7—Create the DirectX events.

```
'Create Devices and Set the DirectX8Events
Private Sub Form_Load()
    CreateDevices                       'Create DirectSound devices
    SetEvents                           'Set up DirectX events
End Sub

'Shut everything down and close application
Private Sub Form_Unload(Cancel As Integer)

    If Receiving = True Then
        dsb.Stop                        'Stop Playback
        dscb.Stop                       'Stop Capture
    End If

    Dim i As Integer
    For i = 0 To UBound(hEvent)         'Kill DirectX Events
        DoEvents
        If hEvent(i) Then dx.DestroyEvent hEvent(i)
    Next

    Set dx = Nothing                    'Destroy DirectX objects
    Set ds = Nothing
    Set dsc = Nothing
    Set dsb = Nothing
    Set dscb = Nothing

    Unload Me

End Sub
```

Fig 8—Create and destroy the DirectSound Devices and events.

DirectX has finished writing to a block. This is accomplished using the DirectXEvent8. Fig 7 provides the code necessary to set up the two events that occur when the lWrite cursor has reached 50% and 100% of the DirectSoundCaptureBuffer.

We begin by creating the two event handles hEvent(0) and hEvent(1). The code that follows creates a handle for each of the respective events and sets them to trigger after each half of the DirectSoundCaptureBuffer is filled. Finally, we set the number of notification positions to two and pass the name of the EVNT() event handle array to DirectX.

The CreateDevices and SetEvents subroutines should be called from the Form_Load() subroutine. The Form_Unload subroutine must stop capture and playback and destroy all of the DirectX objects before shutting down. The code for loading and unloading is shown in Fig 8.

Starting and Stopping Capture/Playback

Fig 9 illustrates how to start and stop the DirectSoundCaptureBuffer. The dscb.Start DSCBSTART_LOOPING command starts the DirectSoundCaptureBuffer in a continuous circular loop. When it fills the first half of the buffer, it triggers the DirectX Event8 subroutine so that the data can be read, processed and sent to the DirectSoundBuffer. Note that the DirectSoundBuffer has not yet been started since we will quadruple buffer the output to prevent processor loading from causing gaps in the output. The FirstPass flag tells the event to start filling the DirectSoundBuffer for the first time before starting the buffer looping.

Processing the Direct-XEvent8

Once we have started the DirectSoundCaptureBuffer looping, the completion of each block will cause the DirectX Event8 code in Fig 10 to be executed. As we have noted, the events will occur when 50% and 100% of the buffer has been filled with data. Since the buffer is circular, it will begin again at the 0 location when the buffer is full to start the cycle all over again. Given a sampling rate of 44,100 Hz and 2048 samples per capture block, the block rate is calculated to be 44,100/2048 = 21.53 blocks/s or one block every 46.4 ms. Since the quad buffer is filled before starting playback the total delay from input to output is 4×46.4 ms = 185.6 ms.

The DirectX Event8_DXCallback event passes the eventid as a variable. The case statement at the beginning of the code determines from the eventid, which half of the DirectSoundCaptureBuffer has just been filled. With that information, we can calculate the starting address for reading each block from the DirectSoundCaptureBuffer to the inBuffer() array with the dscb.ReadBuffer command. Next, we simply pass the inBuffer() to the external DSP subroutine, which returns the processed data in the outBuffer() array.

Then we calculate the StartAddr and EndAddr for the next write location in the DirectSoundBuffer. Before writing to the buffer, we first check to make sure that we are not writing between the lWrite and lPlay cursors, which will cause portions of the buffer to be overwritten that have already been committed to the output. This will result in noise and distortion in the audio output. If an error occurs, the FirstPass flag is set to true and the pointers are reset to zero so that we flush the DirectSoundBuffer and start over. This effectively performs an automatic reset when the processor is overloaded, typically because of graphics intensive applications running alongside the SDR application.

If there are no overwrite errors, we write the outBuffer() array that was returned from the DSP routine to the next StartAddr to EndAddr in the DirectSoundBuffer. Important note: In the sample code, the DSP subroutine call is commented out and the inBuffer() array is passed directly to the DirectSoundBuffer for testing of the code. When the FirstPass flag is set to True, we capture and write four data blocks before starting playback looping with the .SetCurrentPosition 0 and .Play DSBPLAY_LOOPING commands.

The subroutine calls to StartTimer and StopTimer allow the average computational time of the event loop to be displayed in the immediate window. This is useful in measuring the efficiency of the DSP subroutine code that is called from the event. In normal operation, these subroutine calls should be commented out.

Parsing the Stereo Buffer into I and Q Signals

One more step that is required to use the captured signal in the DSP subroutine is to separate or parse the left and right channel data into the *I* and *Q* signals, respectively. This can be accomplished using the code in Fig 11. In 16-bit stereo, the left and right channels are interleaved in the inBuffer() and outBuffer(). The code simply copies the alternating 16-bit integer values to the RealIn()), (same as *I*) and ImagIn(), (same as *Q*) buffers respectively. Now we are ready to perform the magic of digital signal processing that we will discuss in the next article of the series.

Testing the Driver

To test the driver, connect an audio generator—or any other audio device, such as a receiver—to the line input of the sound card. Be sure to mute line-in on the mixer control panel so that you will not hear the audio directly through the operating system. You can open the mixer by double clicking on the speaker icon in the lower right corner of your Windows screen. It is also accessible through the Control Panel.

Now run the Sound application and press the On button. You should hear the audio playing through the driver. It will be delayed about 185 ms from the incoming audio because of the quadruple buffering. You can turn the mute control on the line-in mixer on and off to test the delay. It should sound like an echo. If so, you know that everything is operating properly.

Coming Up Next

In the next article, we will discuss in detail the DSP code that provides

```
'Turn Capture/Playback On
Private Sub cmdOn_Click()
    dscb.Start DSCBSTART_LOOPING    'Start Capture Looping
    Receiving = True                'Set flag to receive mode
    FirstPass = True                'This is the first pass after
Start
    OutPtr = 0                      'Starts writing to first buffer
End Sub

'Turn Capture/Playback Off
Private Sub cmdOff_Click()
    Receiving = False               'Reset Receiving flag
    FirstPass = False               'Reset FirstPass flag
    dscb.Stop                       'Stop Capture Loop
    dsb.Stop                        'Stop Playback Loop
End Sub
```

Fig 9—Start and stop the capture/playback buffers.

```
'Process the Capture events, call DSP routines, and output to Secondary Play Buffer
Private Sub DirectXEvent8_DXCallback (ByVal eventid As Long)

    StartTimer                          'Save loop start time

    Select Case eventid                 'Determine which Capture Block is ready
        Case hEvent(0)
            InPtr = 0                   'First half of Capture Buffer
        Case hEvent(1)
            InPtr = 1                   'Second half of Capture Buffer
    End Select

    StartAddr = InPtr * CaptureBytes    'Capture buffer starting address

    'Read from DirectX circular Capture Buffer to inBuffer
    dscb.ReadBuffer StartAddr, CaptureBytes, inBuffer(0), DSCBLOCK_DEFAULT

    'DSP Modulation/Demodulation - NOTE: THIS IS WHERE THE DSP CODE IS CALLED
'    DSP inBuffer, outBuffer

    StartAddr = OutPtr * CaptureBytes   'Play buffer starting address
    EndAddr = OutPtr + CaptureBytes - 1 'Play buffer ending address

    With dsb                            'Reference DirectSoundBuffer

            .GetCurrentPosition PlyCurs 'Get current Play position

            'If true the write is overlapping the lWrite cursor due to processor loading
            If PlyCurs.lWrite >= StartAddr _
                And PlyCurs.lWrite <= EndAddr Then
                    FirstPass = True    'Restart play buffer
                    OutPtr = 0
                    StartAddr = 0
            End If

            'If true the write is overlapping the lPlay cursor due to processor loading
            If PlyCurs.lPlay >= StartAddr _
                And PlyCurs.lPlay <= EndAddr Then
                    FirstPass = True    'Restart play buffer
                    OutPtr = 0
                    StartAddr = 0
            End If

        'Write outBuffer to DirectX circular Secondary Buffer. NOTE: writing inBuffer causes
direct pass through. Replace
        'with outBuffer below to when using DSP subroutine for modulation/demodulation
        .WriteBuffer StartAddr, CaptureBytes, inBuffer(0), DSBLOCK_DEFAULT

        OutPtr = IIf(OutPtr >= 3, 0, OutPtr + 1)      'Counts 0 to 3

        If FirstPass = True Then        'On FirstPass wait 4 counts before starting
            Pass = Pass + 1             'the Secondary Play buffer looping at 0
            If Pass = 3 Then            'This puts the Play buffer three Capture cycles
                FirstPass = False       'after the current one
                Pass = 0                'Reset the Pass counter
                .SetCurrentPosition 0   'Set playback position to zero
                .Play DSBPLAY_LOOPING   'Start playback looping
            End If
        End If

    End With

    StopTimer                           'Display average loop time in immediate window

End Sub
```

Fig 10—Process the DirectXEvent8 event. Note that the example code passes the inBuffer() directly to the DirectSoundBuffer without processing. The DSP subroutine call has been commented out for this illustration so that the audio input to the sound card will be passed directly to the audio output with a 185 ms delay. Destroy objects and events on exit.

```
Erase RealIn, ImagIn

    For S = 0 To CAPTURESIZE - 1 Step 2       'Copy I to RealIn and Q to ImagIn
        RealIn(S \ 2) = inBuffer(S)
        ImagIn(S \ 2) = inBuffer(S + 1)
    Next S
```

Fig 11—Code for parsing the stereo inBuffer() into in-phase and quadrature signals. This code must be imbedded into the DSP subroutine.

modulation and demodulation of SSB signals. Included will be source code for implementing ultra-high-performance variable band-pass filtering in the frequency domain, offset baseband IF processing and digital AGC.

Notes

[1]G. Youngblood, AC5OG, "A Software-Defined Radio for the Masses: Part 1," *QEX*, July/Aug 2002, pp 13-21.

[2]Information on both DirectX and Windows Multimedia programming can be accessed on the Microsoft Developer Network (MSDN) Web site at **www.msdn. microsoft.com/library**. To download the DirectX Software Development Kit go to **msdn.microsoft.com/downloads/** and click on "Graphics and Multimedia" in the left-hand navigation window. Next click on "DirectX" and then "DirectX 8.1" (or a later version if available). The DirectX runtime driver may be downloaded from **www. microsoft.com/windows/directx/downloads/default.asp**.

[3]You can download this package from the ARRL Web **www.arrl.org/qexfiles/**. Look for 0902Youngblood.zip. □□